83N20177

Final Report

on NASA Grant No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

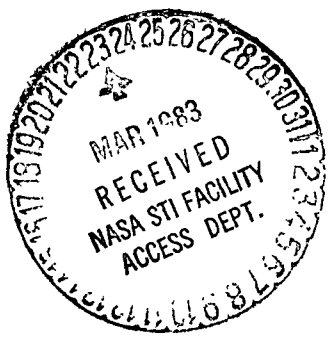
Attention: Edmond H. Senn
ACD

Submitted by:

P. F. Reynolds
Assistant Professor

John C. Knight
Associate Professor

John I. A. Urquhart



Report No. UVA/528213/AMCS83/102

March 1983

DEPARTMENT OF APPLIED MATHEMATICS
AND COMPUTER SCIENCE

Final Report

on NASA Grant No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Edmond H. Senn
ACD

Submitted by:

P. F. Reynolds
Assistant Professor

John C. Knight
Associate Professor

John I. A. Urquhart

Department of Applied Mathematics and Computer Science
RESEARCH LABORATORIES FOR THE ENGINEERING SCIENCES
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528213/AMCS83/102

Copy No. _____

March 1983

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle	5. Report Date
The Implementation and Use of Ada on Distributed Systems with High Reliability Requirements	March 1983
	6. Performing Organization Code 5-28213
7. Author(s)	8. Performing Organization Report No
John C. Knight and John I. A. Urquhart	UVA/528213/AMCS83/102
	10. Work Unit No.
9. Performing Organization Name and Address University of Virginia Thornton Hall Charlottesville, VA 22901	11. Contract or Grant No. NAG-1-260
	13. Type of Report and Period Covered Final Report 03/05/82 - 12/04/82
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665	14 Sponsoring Agency Code

15. Supplementary Notes

None

16. Abstract

This report discusses the issues involved in the use of the programming language Ada* on distributed systems. In particular it is concerned with the effects of Ada programs of hardware failures such as loss of a processor.

It is shown that many Ada language elements are not well suited to this environment. Processor failure can easily lead to difficulties on those processors which remain. As an example, the calling task in a rendezvous may be suspended forever if the processor executing the serving task fails.

A mechanism for detecting failure is proposed and changes to the Ada run-time support system are suggested which avoid most of the difficulties. Ada program structures are defined which allow programs to reconfigure and continue to provide service following processor failure.

* Ada is a trademark of the U.S. Department of Defense.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
distributed systems, Ada, fault tolerance	Unclassified

19 Security Classif. (of this report)	20. Security Classif (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	115	

1. INTRODUCTION

The purpose of this grant is to investigate the use and implementation of Ada (a trade mark of the US Dept. of Defense) in distributed environments in which the hardware components are assumed to be unreliable. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processor they are executing on, and that failures may occur in the underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

We assume that communication between tasks on separate processors will take place using the facilities of the Ada language, primarily the rendezvous. It would be possible to build a separate set of facilities for communication between processors and treat the software on each machine as a separate program. This is pointless however since such

facilities would necessarily duplicate the existing facilities of the rendezvous.

Our work under this grant indicates that the situation is not as good as expected. There seem to be numerous aspects of the language which make its use on a distributed system very difficult. The issues are not raised directly from efforts to implement the language but from the desire to be able to recover, reconfigure, and provide continued service in the presence of hardware failure. It seems that very little attention was paid to these issues in the design of Ada although the language reference manual [1] specifically states that a system consisting of communicating processors with private memories is suitable for executing Ada programs.

Our work under this grant has consisted of:

(1) The preparation of a formal definition of the Ada tasking semantics using the H-graph methodology.

(2) An examination of the language (July 1982, version 9) to thoroughly understand the various facilities and comment on the tasking features.

(3) The generation of a model of the assumed underlying hardware system and the failures of that system which will be tolerated.

(4) Preparation of an initial version of a model of distributed processing. This model will provide the framework for formal discussion of the issues in distributed processing and particularly the issues raised by hardware failure.

(5) An examination of the Ada language with regard to its use and implementation in the assumed environment.

(6) The design of a mechanism for detection of hardware failures and associated signalling facilities to the Ada software.

(7) Development of techniques for writing Ada programs in the assumed unreliable environment to allow for reconfiguration and continued service.

Each of these topics are discussed in the sections below.

2. THE NEED TO COPE WITH HARDWARE FAILURE

The kind of architecture we expect to be in common use for embedded systems in the future in shown in fig 2.1. It is based on the use of a high-performance data bus which links several processors. Each processor is equipped with its own memory. Devices such as displays, sensors, and actuators would be connected to the bus via dedicated microprocessors. Thus these devices would be accessible from each processor.

The bus system would probably be fiber-optic for the various electronic and physical advantages that fiber optics provide. However, the very high data rates and very low error rates make them attractive from a digital point of view also. Digital fiber-optic links are also very inexpensive. For example, a digital transmitter/receiver pair which will operate from DC to 10 Mhz costs less than $500 [2].

The processors in such a system could be very powerful, inexpensive, and with very low power consumption. Processors such as the Motorola M68000 have been shown to outperform a DEC VAX 11/780 on certain problems [3] but are physically small and cost only a few hundred dollars.

A great deal of research has been undertaken in recent years to produce computer architectures of great reliability such as the SIFT [4] and FTMP [5] machines. Why then should there be any concern for software structures which are able to cope with hardware failure? There are several reasons:

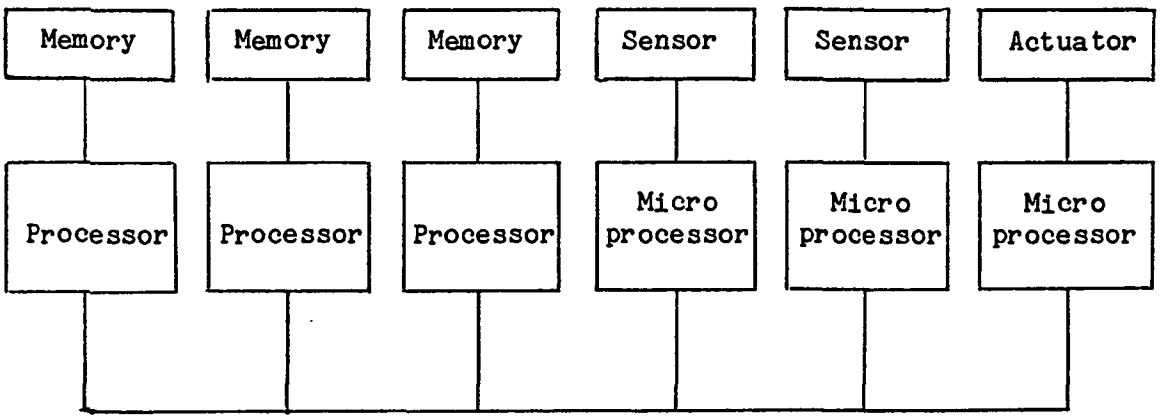


Figure 2.1 - Distributed Architecture

(1) The architectures for highly-reliable systems are very complex and are, in effect, highly-parallel multiprocessors. The reliability is achieved by parallelism. These architectures are the subject of current experimentation and are still unproven.

(2) Even though designed for reliability, these machines may still fail.

(3) Physical damage could cause a processor to fail no matter how carefully the processor was built. Fire, structural failure, excess or unexpected vibration, and so on, could cause enough damage that even a highly-parallel machine would be unable to continue.

(4) Electrical damage from unexpected lightening effects could cause a processor to fail.

(5) In a situation where a major power failure occurred, reserve power might only be provided for some subset of the processors. The switch from full power to limited reserve power might be orderly in which case very sophisticated reconfiguration might be possible. However, it might be preferable to use a single, consistent mechanism for recovery to cope with all cases.

(6) Sophisticated unmanned spacecraft frequently make extensive use of computers but are usually unable to pay the weight and power costs of extensive redundancy (such as in quad redundancy or SIFT). Reconfigurable distributed systems designed to cope with processor or bus failure is an attractive alternative. If the design includes higher processing power than is absolutely needed, and

tasks exist which are not essential to mission success, then some loss of hardware followed by reconfiguration may allow the mission to continue successfully. For spacecraft with extremely long mission times, power and weight limitations again restrict the use of highly parallel architectures though failure is very likely because of the long duration. In such cases carefully selected and hardened components could be used but failure would have to be anticipated. Continued progress and mission success after failure would require reconfiguration after processor failure.

Thus, although great care may be taken with the construction of a digital computer system, failure may still occur. At least with a distributed system there is the possibility that if part of the system was lost, what remained could continue to provide service.

3. FORMAL SEMANTIC DEFINITION OF ADA

In order to be able to implement a language, it is imperative that a precise definition of the language semantics be available. Semantic definitions of programming languages are relatively uncommon because existing methods for semantic definition are difficult to use and, in some ways, inadequate. A semantic definition of Ada was prepared for the Ada Joint Program Office (AJPO) using denotational semantics [6]. This definition is quite difficult to read, but its biggest problem is that the tasking and exception semantics of Ada are totally absent from the definition. The reason is that denotational-semantic methods are not sufficiently powerful to describe tasking.

Previous work at the College of William and Mary produced a semantic definition using H-graph semantics. Since that report, the Ada language has changed substantially and the H-graph definition methodology has been revised considerably [7]. The new definition which we have undertaken is a revision of the earlier work at William and Mary using the latest versions of both Ada and the H-graph method. The current version of the definition was submitted as an appendix to the semi-annual report for this grant, and is included as appendix 1 of this report for completeness.

4. GENERAL EXAMINATION OF THE LANGUAGE

As a part of the activities under this grant, we participated as a volunteer review group for the July 1982 Ada definition. This effort was coordinated by AdaTEC as an attempt to generate comments from the United States about the revised language before the final version (which would be used for the ANSI canvass) was printed. We found the revised language definition document to be very different from the July 1980 version and chose to put all of our effort into reviewing the tasking definition thoroughly rather than reviewing the entire language definition superficially.

The result of our review was a set of 35 questions which were discussed at the meeting of the volunteer reviewers at the AdaTEC conference in Boston (June 1982). Our comments were well received and many were found to have substantial content. These were to be passed on to the Ada design team for consideration. An examination of the July 1982 Ada reference manual (denoted version 16) indicates that our comments were either not received in time or were not acted upon since most of the language difficulties still exist. The problems also exist in ANSI Ada. A copy of our questions was included in the semi-annual report for this grant, and is included as appendix 2 of this report for completeness.

In general our concerns about Ada are to do with time. Some examples are:

(1) The conditional entry call is defined in terms of the word "immediately" but does not define "immediately", and so we have no way of

knowing how the call is supposed to be implemented. There are several ways which are entirely different based on the current language reference manual definition (see section 7 for details).

(2) The timed entry call does not define when the time for the call is to begin. There are again several different interpretations. Worse however, is the fact that the timed entry call does not provide a useful facility for the programmer in its current form given any definition. The problem which it should address is the need to be able to time-out easily after a rendezvous has begun rather than before.

These issues demonstrate why a formal definition of Ada is so important. We cannot decide exactly what is supposed to be implemented given the current language definition which is in English and therefore ambiguous, imprecise, and incomplete.

Unfortunately, the issues with the conditional and timed entry calls, and with other language elements, are actually much more serious in a distributed system. They are precisely the tools which the programmer must use to communicate between tasks on different processors. It is this kind of communication which concerns us most in the context of hardware failure, and it is in this area where the language seems to have major weaknesses.

5. UNDERLYING HARDWARE MODEL

Initially, we assume that communication between processors on a distributed system will be implemented using layers of software that conform for the most part to the ISO standard seven-layer Reference Model [8]. The hardware topology that is used for a distributed system need have very little impact on the programming of the system at the application-layer level. In principle, provided the implementation knows how tasks are distributed to processors and how communication is to be achieved, the various tasks can synchronize and communicate at will with no knowledge of their location.

To discuss implementation and recovery in the context here, we found it necessary to have an underlying hardware model. Our model assumes that a set of processors are connected to some sort of communications bus system which we do not define. The bus system could be a ring, multiple rings, a crossbar switch, etc. Peripheral equipment such as sensors and actuators are also assumed connected to the bus system, and the connection is assumed to be provided by a microprocessor dedicated to the interface.

The kinds of hardware failure that we are concerned with are not addressed by the ISO protocol. The ISO protocol is concerned with communications failures such as dropped bits caused by noise, loss of messages or parts of messages, etc. Also, situations such as a processor "slowing down" or incorrectly computing results are not of interest here (though they are important nevertheless). We assume that such events are taken care of by hardware checking within the processor. The only

class of faults not dealt with elsewhere is the total loss of a processor or bus with no warning. These are the difficulties we will attempt to deal with.

We feel that this hardware model and associated failure model is directly relevant to avionics and other embedded systems, and are also relevant to spacecraft systems. As noted above, spacecraft unable to pay the weight and power costs of extensive parallelism to provide reliability could use reconfigurable distributed systems designed to cope with processor or bus failure as a possible alternative. Some loss of hardware followed by reconfiguration may allow the mission to continue successfully.

6. A CONCEPTUAL BASIS FOR DISTRIBUTED PROGRAMMING

A distributed program will, in general, be non-deterministic. Further, the execution path of the program can depend on processor speeds and scheduling algorithms, which cannot be specified in the program. Under these circumstances, it is not clear what a program means. In this section a program is considered to define a set of possible execution sequences. An assignment of tasks to processors along with constraints on processor speeds and scheduling will be called a realization. A proper realization of a program will define a subset of the set of execution sequences defined by the program. This model is compared to more conventional models; the differences are seen to lie in the treatment of real-time issues and crashes. Some general conclusions are drawn, the most important being that in order to be truly distributed a program must be able to handle crashes, and finally areas needing further research are listed.

6.1. A PROGRAM DEFINES A SET OF EXECUTION SEQUENCES

The idea of a program defining a set of execution sequences is due to Pnueli, and has been used by Owicki and Lamport to apply temporal logic to proofs of liveness. Their ideas are extended to deal with the notion of time, and the semantics of Ada. Time dependent actions, crashes, and priorities, are then discussed in more detail.

6.1.1. DEFINITION

Owicki and Lamport define a program state to consist of an assignment to each program variable and a control component. The control com-

ponent is a set of atomic actions which are ready for execution; the next state is reached by choosing some action in the control component and executing it. An execution sequence is a sequence of program states obtained in this way from the initial state.

While adequate for their purposes, Owicki and Lamport's model must be extended in order to deal with the semantics of Ada. In Ada an action can be executed or not depending on whether some other action has occurred, or on time. An accept waiting for an entry call, for example, is half ready. Half-ready actions lead to the idea of giving actions in the ready list guards and executing them only when the guards are true. Once you have guards there is no need for the ready list and the program can be regarded as a set of guarded actions. An execution sequence for the program would be a set of states obtained by executing the following loop:

```
loop
   evaluate guards;
   execute some action with a true guard;
   exit when terminate_program;
end loop;
```

This still does not deal with the problem of time-dependent actions. A guard can refer to time but as yet there is no concept of time in the execution. Putting it there raises difficulties since it will then matter whether actions are performed serially or concurrently.

A guarded action will now be defined to consist of a guard, an action, and an execution time. The program state will be extended to

include a variable CLOCK, and to model concurrent execution of actions, the execution loop will execute any subset of the set of actions with true guards, and then increment CLOCK by the largest execution time of the set of actions executed. Thus, the execution loop becomes:

```
loop
  evaluate guards;
  execute some subset of the set of actions with true guards;
  exit when terminate_program;
  increment CLOCK by max{execution-time of action executed, 1};
end loop;
```

The subset of the set of actions that is executed may be the empty set; in this case the clock in incremented by one tick. There is no guarantee that an action will not have a true guard, be ignored, and then have the guard become false. However if an action has a true guard, which remains true, the action will eventually be executed.

The execution times of actions are intended to define allowable execution sequences; an implementation may have quite different timings so long as only allowable execution sequences will result.

6.1.2. PRIORITIES

These only make sense if tasks are running on the same processor, however most languages to not allow a program to specify the assignment of tasks to processors. Without knowing which tasks will be running on a particular processor, the use of priorities is rather a crude tool. A bad situation gets worse when priorities are static and assignment of

tasks to processors is dynamic, as it must be to reconfigure after a crash.

Priorities are thus better removed from the program to the realization, to be treated along with other scheduling constraints.

6.1.3. CRASHES

The effect of a crash is to remove some set of actions from the program. This changes the program, and will usually lead to an illegal execution sequence. On a distributed system, the system can be reconfigured after a crash if, at a minimum, it is known crash has happened, and it is known which tasks were on the crashed processor. In order to restart tasks it will usually be necessary to have a consistent set of data available to a non-crashed processor.

6.2. REALIZATION OF A PROGRAM

A program defines a set of allowable execution sequences. The execution of a program can take place in many different ways; the only constraint is that the actual execution sequence be allowable. A realization of a program specifies the number of processors to be used, how tasks will be assigned to processors, constraints on processor speeds, and scheduling algorithms.

A proper realization restricts execution sequences to a subset of the set of execution sequences defined by the program.

A strict realization restricts execution to a single execution sequence.

6.3. TWO MODELS COMPARED

The model described above of a program as a set of allowable execution sequences along with a realization is compared with the conventional model of a distributed program--a set of tasks running on virtual processors. The models differ when programs depend on timing.

6.3.1. VIRTUAL PROCESSOR MODEL

A distributed program can be viewed as a collection of tasks each running on its own (virtual) processor. Tasks proceed independently except at synchronization points; processor speeds can be arbitrary, except that none can be infinitely slow. If the processors are virtual, scheduling appears to vary processor speed.

Any execution sequence possible under this model is acceptable. In particular any variation in processor speeds leads to an acceptable execution sequence. Real time programs are thus excluded from consideration, since no real time program could run arbitrarily slowly and give acceptable results.

Crashes and reconfiguration are also excluded from this model, although in this case, the model can be extended. A crash can be considered as several processors stopping at the same time; reconfiguration consists in starting new tasks on new virtual processors.

6.4. SOME IMPLICATIONS OF THE MODEL

From the perspective of execution sequences and realizations some general conclusions about distributed programs can be drawn.

6.4.1. CRASH RECOVERY IS NECESSARY

If tasks are running on several processors and one of the processors crashes, then if no reconfiguration takes place, the remaining tasks will run until they terminate or reach a synchronization point with a task which was running on the processor which crashed. This execution sequence could have been obtained from a uniprocessor, which crashed at the same point as the crashing processor in the multiprocessor case. The point is, work that was done after the crash in the multiprocessor case could have been scheduled earlier and completed before the crash in the uniprocessor case.

The situation is completely different if a multi-processor system can reconfigure and provide service after a crash. Now the execution sequence could not be provided by a crashing uniprocessor.

Thus a distributed system must be able to reconfigure, if it is to do better than a uniprocessor, when a crash occurs. This has several important consequences:

(1) There must be knowledge of the crash.

(2) There must be knowledge of which tasks have been lost.

(3) There must be a consistent set of data (it may be out-of-date), so that tasks can be restarted.

(4) In order to know where to restart tasks a processor/task map must exist in each processor.

6.4.2. CLASSIFICATION OF PROGRAMS

In this section crashes will be assumed to be impossible. Realization independent programs.

These are programs whose execution paths are not affected by the realization under which they are being executed. That is,

(R) ({ execution sequences under realization R } = { execution sequences defined by the program })

Time independent programs

These are programs whose execution paths may be affected by the realization under which they are being executed, but every realization is a proper realization.

(R) ({ execution sequences under realization R } <= { execution sequences defined by the program })

Time dependent programs

These are programs for which not all realizations are proper.

6.4.3. PREFERRED EXECUTION SEQUENCES

Often a program is written so that it can take care of abnormal conditions. If this is done by providing degraded service then a realization that ensured that the abnormal condition always occurred would always provide degraded service. Thus such a realization while proper would be less than ideal. An example:

Suppose a task makes an entry call to a server-task, whenever it needs to know the current position. If the server is so busy that is cannot provide the current position quickly enough the task can cancel the entry call and use an estimate for the current position.

```
select
   SERVER.GET_POSITION(POSITION : in coordinate_triple);
   -- do computation
or
   delay MAX_WAIT;
   -- do computation using an estimate of the position
end_select;
```

Now, a realization in which the server never provided the current position could still be a proper realization. On the other hand, if the execution sequence excluded the calculation using the estimate of the position, proper realizations would have to guarantee that the server could always calculate the position in time.

So it seems that a proper realization must have a further condition imposed on it, a condition giving probabilities that certain sets of execution sequences will be followed when the program is executed under the realization. This is not as unpleasant as it seems, since many of the properties of a realization will also be given in term of probabilities, and it does not seem unreasonable to consider a realization satisfactory, if the program can deal with the worst case, and the worst case only happens rarely.

7. ADA ISSUES AND DIFFICULTIES

In this section difficulties with the use of Ada on a distributed system, and with the semantics of Ada, are described. Proposed solutions to some of the problems raised here are given in section 9.

7.1. RENDEZVOUS

The rendezvous is the fundamental way for tasks to communicate. Even in the simplest case difficulties arise when processor failures are taken into account. Timed and conditional entry calls, intended to provide solutions for some of these difficulties, are seen to raise difficulties of their own, both in the meaning of their semantics and in their use.

7.1.1. SIMPLE RENDEZVOUS

A simple rendezvous in Ada consists of a calling task C making an entry call, S.E, to a serving task S, which contains an accept statement for the entry E. The syntax is shown in figure 7.1. The semantics of the language require that if the call is made by C before the accept is

```
Calling Task C                  Serving Task S

      .                           ACCEPT E DO
      .                               .
      S.E;                            .
      .                               .
      .                           END E;
```

Figure 7.1 - The Syntax Of A Simple Rendezvous.

reached by S, C is suspended until the accept is reached. If S reaches the accept before the call is made by C, S is suspended until the call is made. In either case, C remains suspended until the rendezvous itself is complete.

In order to look at the issues arising from a rendezvous (in particular, the effect of processor failures) it is necessary to specify an implementation of the rendezvous at the message passing level. After the possibilities for processor failures during a rendezvous are discussed, some general conclusions are drawn about the rendezvous mechanism.

7.1.1.1. IMPLEMENTATION OF THE SIMPLE RENDEZVOUS

Only the simple case of a task C calling an entry E in a serving task S will be considered. Further, it will be assumed that the call is made before S has reached the corresponding accept; the case where the server waits at its accept is similar. The messages that would be needed are shown on figure 7.2.

The calling task C asks to be put onto the queue for entry E. When S reaches its accept for E, it sees that C is on the queue. At this point S checks to see if C has been aborted. When the CHECK_CALLER message arrives at C, C can be considered to be engaged in the rendezvous. When the reply reaches S, S will start to execute the rendezvous code. When it is completed the RENDEZVOUS_COMPLETED message would awaken C which would continue.

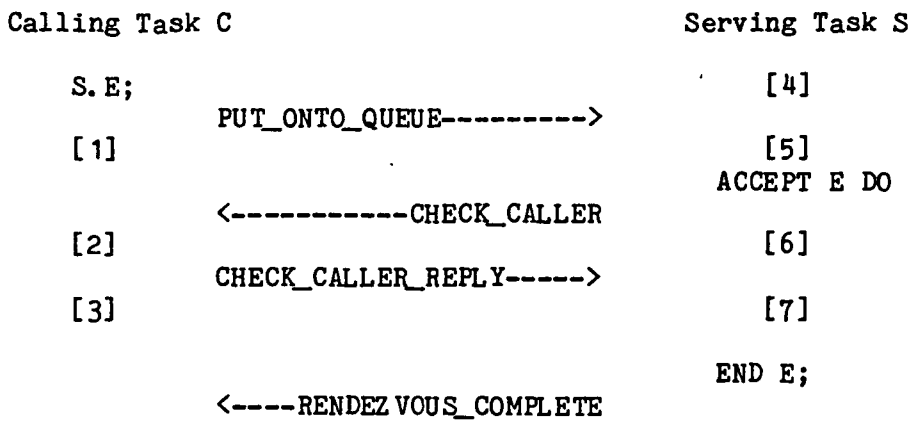


Figure 7.2 - The Messages Used To Implement The Rendezvous

Note that all messages are assumed to arrive safely.

7.1.1.2. THE EFFECTS OF PROCESSOR FAILURE ON SIMPLE RENDEZVOUS

Using the implementation of a simple entry call shown above, what happens if either the server or the calling task crashes? There are seven cases of interest and they are discussed below. The numbers refer to figure 7.2.

CALLER CRASHES AT: — EFFECT ON SERVER

[1] The message CHECK_CALLER will not be able to arrive. The effect on the sender should be equivalent to a negative reply to the CHECK_CALLER message (e.g. if the caller had been aborted, but not yet removed from the queue). That is the server would remove the caller from the queue and remain waiting at the accept.

[2] The message CHECK_CALLER arrives, then the caller crashes and the reply is never sent. If the server cannot find out that there has been a crash, the server will be trapped waiting for the message CHECK_CALLER_reply.

[3] When the caller crashes during the rendezvous, the situation is similar to the case where the caller is aborted during the rendezvous. In both cases the server can

continue. At the end of the rendezvous the RENDEZVOUS_COMPLETE message cannot arrive; as before, if the server can detect that there has been a crash, the server can continue.

SERVER CRASHES AT:	EFFECT ON CALLER
[4]	The message PUT_ONTO_QUEUE cannot arrive. The situation is similar to the case where the server is abnormal.
[5]	Here the caller is on an entry queue when the server crashes. As before if the crash cannot be detected the caller will be trapped.
[6]	The message CHECK_CALLER has arrived at the caller who now considers that the rendezvous has started; the reply cannot arrive. Again without crash detection the caller will be trapped. (Note that even if the caller were using a timed entry call, the timer would have been turned off by the message CHECK_CALLER.)
[7]	The server crashes during the rendezvous. (Timed and conditional entry calls give no protection as they time the delay to the start of the rendezvous.) Again the caller is trapped unless the crash can be detected.

The server task is is not seriously affected when the calling task crashes. At worst, time is lost doing work for a task that is not there to receive it. The calling task is in a much worse situation when the server crashes. If the rendezvous has not already started the caller will wait on the entry queue for ever. Timed entry calls (discussed below) can handle this situation, if they are implemented by having the caller task do the timing. If the server task crashes after the rendezvous has started, even a caller who has made a timed or conditional entry call will be trapped for ever.

7.1.1.3. FURTHER ISSUES WITH THE SIMPLE RENDEZVOUS

What the caller would like to have, and what even timed and conditional entry calls do not give, is a guarantee that after a certain time

it will be possible to proceed. The rules of the language imply that once a rendezvous has started the caller cannot withdraw until it is completed. Clearly withdrawal is necessary when the server crashes; even when the server has not crashed the caller may wish to withdraw and take some alternative action. As this facility must be provided to deal with server crashes it might as well be provided in all cases.

Thus we suggest that the caller should be able to withdraw from a rendezvous at any time. Clearly this violates the semantics of the rendezvous as presently defined since the server may be depending on the fact that the caller is suspended.

The situation is not this simple however. It is not possible to dictate that this is a violation of the language semantics. Suppose that a rendezvous is in progress and that the server calls a third task. This third task may also be relying on the original caller's suspension. If the original server crashes, the third task called by the original server may still depend on the original caller's suspension. The original caller may proceed however because its server no longer exists.

This raises another fundamental issue about the semantics of the rendezvous. If the server task cannot depend on knowing that a particular task is suspended during a rendezvous, is there any reason why the caller should be suspended?

7.1.2. RENDEZVOUS BY TIMED ENTRY CALL

Timed entry calls are intended to solve some of the problems raised above. In fact, they raise further problems about their meaning

and their implementation.

The semantics of the timed entry call appears to be quite straight-forward:

> A timed entry call issues an entry call that is cancelled if a rendezvous is not started within a given delay.

In a distributed system, however, messages will take time to get from a task on one processor to a task on another. Even if the underlying message passing system can guarantee that a message will eventually arrive correctly, this will be implemented at a lower level by a protocol which may well involve acknowledgement of messages, and the resending of messages that have been lost. A message can certainly be delayed for some arbitrary length of time. Even physical separation of the processors may impose a significant delay.

One possible interpretation of the timed entry call would be to count the total time until the rendezvous is started. Message passing time and time on the entry queue would be included. This interpretation has to be ruled out by the statement in the language definition that a timed entry call with a delay of zero is the same as a conditional entry call.

> If a rendezvous can be started within the specified duration (or *immediately, as for a conditional entry call, for a negative or* zero delay), it is performed and the optional sequence of statements after the call is then executed.

If the delay included both message passing time and time on the queue, a delay of zero would be impossible and a timed entry call with a delay of

zero would never succeed.

The only other interpretation of the delay is that it is just the delay on the entry queue. We have to assume that the delay intended by the language definition is waiting time on the queue since this has a meaning when the specified delay is zero.

7.1.2.1. IMPLEMENTATION AND ISSUES WITH TIMED ENTRY CALLS

Once it has been decided that the delay means waiting time on the entry queue, the important implementation question becomes "who is to do the timing". The calling task cannot do the timing. It is impossible for it to measure waiting time on the entry queue accurately since the message passing time can vary. Thus it is essential that the serving task does the timing. However, if the serving task does the timing and then crashes, what will happen to the calling task?

A timed entry call gives protection against having to wait too long on the entry queue, however what the task issuing the call needs is some guarantee that it will not be trapped in an attempt to communicate, and forced to miss a deadline. It does not matter to the task, whether the time is spent waiting on a queue, or attempting to send a message.

If the timed entry call is implemented by having the server do the timing and the server crashes before a rendezvous is started the caller will be trapped. Even if there is no crash the calling task must wait for a message from the server. If the server is doing the timing, that message may need to be re-sent several times, so the calling task may have to wait an arbitrary time .

If the calling task were able to do the timing then an infinite wait could be avoided when the server crashed. As we have noted however, this method of timing is unrealistic.

We conclude that there are many issues with the timed entry call. It does not provide the kind of prevention that is desirable, the semantics are unclear, and it is very difficult to implement. An analysis of the message traffic necessary for the timed entry call can be performed that is similar to that shown in figure 7.2. The issues which arise when considering failure are similar but more extensive than the simple rendezvous.

7.1.3. RENDEZVOUS BY CONDITIONAL ENTRY CALLS

The semantics of the conditional entry call appear to be quite straight-forward:

> A conditional entry call issues an entry call that is then cancelled if a rendezvous is not immediately possible.

By a similar argument to that used with timed entry calls, we conclude from the rules of the language that "immediately" must mean zero waiting time on the entry queue. As message passing time can vary, "immediately" may turn out to be an arbitrary delay. Unless there is an upper bound on the time a message takes, conditional entry calls cannot be used by a task to ensure that a rendezvous will be started within a given time. The use of a conditional entry call is thus restricted to the case where a caller task wishs to make an entry call only when the server is in a certain state (i.e. able to accept the entry). We feel

that this makes the conditional entry call virtually useless.

7.2. SUBTASKS

In this section, we describe task definition and activation of nested tasks in detail. Task creation by allocators will not be considered. The rules given in Ada for exceptions and aborts during these processes are described, and we show the effects of processor failures on these rules.

7.2.1. SUBTASK DEFINITION AND ACTIVATION

A task is started in two steps. First it is created, at this point entry calls can be made to it, then it is activated, that is, the declarative part of the body is elaborated. Creation occurs when the task-object declaration is elaborated, this happens in the declarative part of the parent unit; activation occurs after the declarative part of the parent unit, between the BEGIN and the first statement of the body.

Figure 7.3 shows an example of the syntax of two nested tasks. The parenthesized numbers in the figure show points where various problems can occur and where the language defines the consequences of these problems. The problems are:

AT	ACTION	RESULT
[1]	Exception in P	A and B become terminated [RM.9.3.(4)]
[1]	P aborted	A and B become abnormal and thus, because their activations have not started, terminated. [RM.9.10.(4)] [RM.9.10.(5)]
[2]	Exception in P	No exception can be raised in P at this point. If A has completed its activation, an exception raised in A is not propagated to P, if an

```
     -- These tasks are inside parent P.
     task type T is
     .
     end T;

     A : T; -- task A is created on processor P1.
     B : T; -- task B is created on processor P2.
[1]
          .
          .
     -- End of the declarative part of parent.

     begin
     -- A message is sent to P1 to 'activate A'
[2]
     -- a message is sent to P2 to 'activate B'
     -- the activations of A and B are done in
     -- parallel on processors P1 and P2.
     -- After notifying P that their activation
     -- is complete, A or B can continue.
     -- P waits until the activations of both A
     -- and B are complete, and then continues
     -- with the first statement in its body.
[3]
                - - - - - - - -

     task A goes through the following stages:
          created
[4]
          activation started
[5]       .
          activation completed
          start to execute body
[6]
```

Figure 7.3 - Nested Tasks.

exception is raised during the activation of A, A becomes completed and the exception TASKING_ERROR is raised in P when the activation of B is finished. (at 3) [RM.9.3.(3)]

[2] P aborted — A has started its activation and becomes abnormal, B has not started its activation and becomes terminated as above. [RM.9.10.(4)] [RM.9.10.(5)]

[3] Exception in P	This is not propagated to A or B. If P is a subprogram or a block the exception will not be propagated until A and B are both terminated. [RM.11.5.(8)] [RM.11.5.(9)]
[3] P is aborted	A and B become abnormal since they are dependent on P. [RM.9.10.(4)]
[4] A is aborted	A becomes terminated. P cannot proceed since the activation of A cannot be completed. [RM.9.3.(2)] [RM.9.10.(4)] [RM.9.10.(5)]
[5] Exception in A	A becomes completed. TASKING_ERROR is raised in P at [3]. [RM.9.3.(3)]
[5] A is aborted	A becomes abnormal. P cannot proceed since the activation of A cannot be completed. [RM.9.3.(2)] [RM.9.10.(4)] [RM.9.10.(5)]
[6] Exception in A	The exception is not propagated to P. [RM.11.5.(8)]
[6] A is aborted	A becomes abnormal. No direct effect on B or P. [RM.9.10.(4)]

Consider now the effects of processor failure on the definition and activation of subtasks:

P crashes at [1]
A and B are created and can accept entry calls. The callers are then trapped, since A and B will never be activated.

P crashes at [2]
A can continue with its activation. B is created and can accept entry calls. The callers are then trapped since B will never be activated.

P crashes at [3]
Similar to the previous case except that A and B can both continue. Callers will not be trapped.

A crashes at [4] or at [5]
P will be trapped, waiting for the completion of A's activation.

A crashes at [6]
This will have no effect direct effect on either P or B unless they try to communicate with A.

There are many difficulties here and they are quite substantial.

We conclude that nested tasks in an Ada program present very serious

problems on a distributed system where processor failure may occur.

7.3. GLOBAL VARIABLES

Global variables certainly exist in the syntax of Ada. If a task uses a non-local variable which happens to belong to a task running on a separate processor, a naive implementation could generate a great deal of message passing. It would appear therefore that the same kinds of difficulties that exist with the simple rendezvous occur with global variables. Access to a global variable on another processor requires that a dialogue take place, and failure of the processor on which the global variable resides could trap the task attempting to reference the variable.

However the language allows an implementation to use a copy of the non-local variable, updating it only at synchronization points. A global variable becomes a local variable with implied update messages at synchronization points. Where several tasks are constrained to always remain together on a single processor, globals exist in the usual sense; the part of the program consisting of those tasks is, of course, not distributed.

The language definition in the area of updating shared variables has changed several times as Ada has been revised. We have not had an opportunity to review the definition contained in the Ada standard with sufficient care that we can be certain that there are no problems with the use of global variables in ANSI Ada.

7.4. PROGRAM STRUCTURE FOR A DISTRIBUTED SYSTEM

An Ada program will have a tree structure, with the main program at the root. A set of distributed processors will in general not have a tree structure. How then should the structures be related? This is a remarkable disparity. In general, program structure and the organization of the hardware need not be related. However, the fact that a system is distributed is so central to its operation that ignoring this in program structure seems a mistake. Ada programs have the conventional nested format that derives from Algol 60, and this forces a program structure which cannot be elegantly mapped to the hardware structure.

Despite this disparity, Ada tasks have to be mapped onto processors. Which tasks should be assigned to which processors? Several objectives seem reasonable, such as:

(1) Minimize inter-processor communication.

(2) Maximize computing power by distributing tasks so that processors are evenly loaded.

(3) Make reconfiguration and continued service in the face of processor failures feasible.

Combining these issues with those described in the section above on subtasks leads us to various conclusions about program structure which are discussed in section 9.

7.5. REALIZATION CONTROL

In the model proposed in section 6, a program defines a set of allowable execution sequences. A realization specifies assignment of tasks to processors, processor speeds, and scheduling algorithms. How should a realization be specified and where should it be specified?

In Ada, static priorities are part of the language. They are intended to provide constraints on a scheduler. Clearly they are not sufficient in a distributed system that is trying to reconfigure, since a task with a high priority may be unimportant when a lower level of service is being provided after a processor has failed. It seems to us that some form of dynamic priorities is needed.

A distributed system must be able to reconfigure after a processor failure. This implies that tasks must be able to be dynamically assigned to processors. The program which specifies how this is to be done will be in important part of any reliable real-time system. It is not clear whether such a program could be written in Ada, an extended Ada, or even what primitives the language should contain.

8. FAILURE DETECTION AND SIGNALING

Processor failure cannot be dealt with unless it can be detected. Details of the failure must also be supplied to the software which is to cope with the reconfiguration. How can Ada programs detect hardware failure and what information is needed for reconfiguration? In this section, we present an approach to hardware failure detection and the rational for its choice.

8.1. FAILURE DETECTION

Failure detection could be performed by hardware facilities over and above those provided for normal system operation. Alternatively, failure could be detected by system software. The hardware option is less desirable because it requires additions to existing or planned systems and the detection hardware itself could fail. We suggest therefore the use of software failure detection.

Software failure detection can be either passive or active. A passive system might rely on tasks assuming that failure had occurred if actions did not take place within a "reasonable" period of time. We will refer to this as timing out. Alternatively, a passive system could require that all messages passed between tasks on separate processors be routinely acknowledged. Thus the sender would be sure that the receiver had the message and presumably would act on it. Note that this is a particularly simple case of timing out since failure has to be assumed if no acknowledgement is received.

The disadvantages of passive detection are:

(1) Timing out assumes an agreed-upon upper limit for response time.

(2) A failed processor will not be detected until communication is attempted and this may be long after the failure has occurred.

Upper bounds on response time may be hard to determine. Very complex situations can arise from an incorrect choice. The reason for a lack of response from a task on another processor may not be failure of that processor but merely a temporary rise in its workload. The consequences could be an assumption by one processor that another had failed, followed by reconfiguration to cope with the loss. Clearly, if this assumption is wrong, two processors could begin trying to provide the same service.

Being unaware that a processor had failed will lead to a loss of the service it was providing until the failure is noticed. In a system with many processors each providing relatively few services, the amount of inter-processor communication might be quite low. A failed processor may go unnoticed for a sufficient time that physical damage might result from its lack of service.

It is for these reasons that we reject passive software failure detection and suggest the use of active software failure detection. In an active system, some kind of inter-processor activity is required "periodically" and if it ceases, failure is assumed. The messages which are passed are usually referred to as heartbeats. Heartbeats can take two basic forms. In the first, each processor broadcasts a heartbeat

periodically and all other processors monitor its presence. In the second, a message is passed sequentially from processor to processor and its non-arrival at the appointed time is a signal that one processor in the system has failed.

A final question of implementation is whether the generation and monitoring of heartbeats should be the responsibility of the programmer or the Ada run-time support system. We favor the run-time support system for reasons discussed below.

8.2. FAILURE SIGNALLING

As soon as a heartbeat disappears, the remaining processors in the system will be aware that a failure has occurred and they will know which processor has failed. This information must be transmitted to the software running on each remaining processor so that reconfiguration can take place. The information is available to the run-time support software in some internal format, but how should it be transmitted to the Ada software?

One approach is to use the languages exception mechanism, and for the run-time system to generate an exception on each processor. Another approach is to view the required signal as being very like an interrupt, and transmit the information to the Ada software in the way that interrupts are transmitted; namely by an entry call. We prefer this latter approach because it can take place in parallel with any activity that might already be going on. A task designed to cope with reconfiguration could be present on each processor and suspended at an accept statement for the entry which will be called when a failure occurs. This allows

each processor to have a "focal point" for reconfiguration. If exceptions are used, the correct placement of the necessary handler is difficult to determine because it will be impossible to know what tasks will be engaged in what activities when the exception is generated.

Thus we propose that a special task be defined on each processor which will contain an entry with a single parameter. The parameter will be of some scalar type which conveys for a given call which processor in the system has failed. This task will be normally suspended on the accept statement for the special entry so that when a failure occurs, an entry call with the appropriate parameter will be generated. The task will then be activated and will contain code following the accept statement to handle reconfiguration.

8.3. EXISTING RENDEZVOUS TERMINATION

It is not sufficient to detect failure and inform the software of the failure using the methods described above. As discussed in section 7, the Ada rendezvous can lead to situations in which a calling task is permanently suspended if the processor on which the server is executing fails. These tasks which would be permanently suspended must somehow be released.

The mechanism which we propose to cope with this situation is shown in figure 8.1. Whenever a rendezvous takes place between tasks on different processors, the run-time support system on the processor executing the caller records the details of the rendezvous in a message log. Whenever a failure is detected, each processor checks its message log to see if any of its tasks would be permanently suspended by the failure.

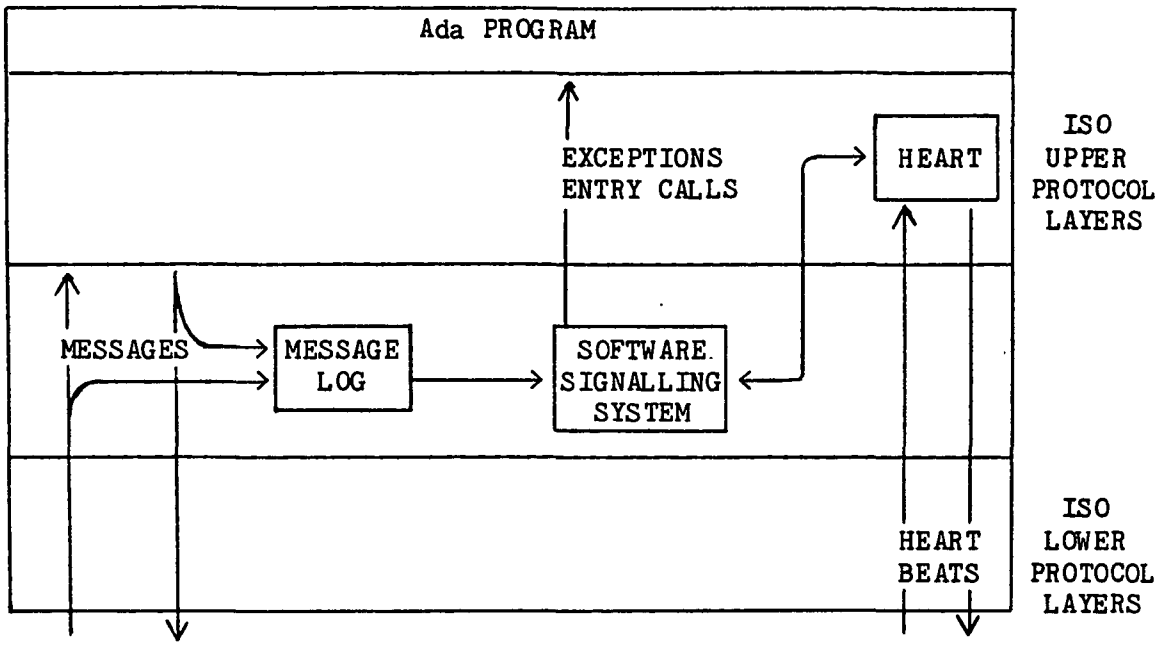


Figure 8.1 - Implementation Model

If any are found, they are sent "fake" messages. They are called fake because they are constructed to appear to come from the failed processor but clearly do not. The message content is usually equivalent to that which would be received if the serving task had been aborted. In this way, each processor is able to ensure that none of its tasks is permanently suspended. However, it is the responsibility of the tasks themselves to ensure that their subsequent actions are appropriate.

There are circumstances in which it is not desirable to allow the calling task to proceed. In these cases, the fake message will be sent to the run-time support system rather than the task itself. The fake message will indicate the kind of action that the run-time system should take; such as abort the task or generate an exception.

Clearly it is possible for unsuspecting tasks to attempt to rendezvous with tasks on the failed processor after failure has been detected, signaled, and other rendezvous aborted. This situation can be dealt with easily if the run-time support system returns a fake message immediately indicating that the serving task has been aborted and that rendezvous is not possible.

Because of the fact that a fairly extensive set of facilities is required in the run-time system for fake messages, we suggest that the heartbeats be handled here also. There is a clear need for cooperation between the heartbeat monitoring system and the fake message system. Operating both at the same level is probably the only practical approach. This has the additional advantage that the programmer is not burdened with the need to include the heatbeat system in his program. Finally, the heartbeat system is so central to the reliability of the entire system that it should operate at the lowest practical level of the software system. Thus it relies for its operation on the correct operation of the minimum amount of other software.

9. FAILURE TREATMENT IN Ada

When a processor fails in a distributed system, the system must respond so that it can continue to provide service. This requires several steps:

(1) The failure must be made known to the software which remains.

(2) Tasks which are suspended indefinitely in rendezvous as a result of the failure must be freed or aborted.

(3) Tasks which are suspended indefinitely as a result of failure occurring during the subtask activation process must be freed or aborted.

(4) A reconfiguration strategy must be chosen.

(5) Substitute tasks must be started.

(6) The substitute tasks must be provided with data consistent across the processors.

In this section we discuss these various issues.

9.1. FAILURE DETECTION

We assume the existence of the the heartbeat mechanism described in section 8. This will allow all operating processors to be aware of processors that have failed, and there will be an upper bound on the delay between failure and detection. Further, we assume that once detected, failure can be signalled to the software using either entry calls, exception generation, or other fake messages as necessary. Thus, failure

detection will be taken care of totally by the support system, and the software will be informed using existing facilities of the language.

9.2. AVOIDING ENTRAPMENT DURING RENDEZVOUS

Recall from section 7 that if the processor running the calling task in a rendezvous fails, the serving task can continue, complete the rendezvous and continue. However, if the processor running the serving task fails, the calling task is trapped. In fact, all tasks waiting on the server's entry queues are trapped.

Recall from section 8 that each processor will have information in its message log showing which of its tasks were either engaged in rendezvous or waiting on entry queues of tasks on the failed processor. Thus entrapment can be prevented by returning suitable fake messages to these tasks. The only question is what fake message should be sent?

The situation is very similar to that which arises when the serving task is aborted. One possibility therefore is to force a tasking error in the calling task. We feel that this is probably adequate. However, we suggest that an alternative worth considering is the definition of a new exception (which we call PROCESSOR_FAILURE) which would be raised under these circumstances.

The reason for not using the tasking error exception is the fact that the cause of the exception would be clearer. If tasking error is used, the task receiving the exception might not realize the cause of the difficulty and might attempt to rendezvous again with a task on the failed processor. This is always possible and would be dealt with but

it is preferable to avoid it if possible.

9.3. AVOIDING ENTRAPMENT DURING TASK ACTIVATION

In section 7.2, problems arising from failures during the activation of tasks were described. In this section, solutions to these prob-

```
        -- These task definitions are inside parent P.
        task type T is
        .
        end T;

        A : T; -- task A is created on processor P1.
        B : T; -- task B is created on processor P2.
[1]
             .
             .
        -- End of the declarative part of parent.

        begin
        -- A message is sent to P1 to 'activate A'
[2]
        -- a message is sent to P2 to 'activate B'
        -- the activations of A and B are done in
        -- parallel on processors P1 and P2.
        -- After notifying P that their activation
        -- is complete, A or B can continue.
        -- P waits until the activations of both A
        -- and B are complete, and then continues
        -- with the first statement in its body.
[3]
                  - - - - - - - -

        task A goes through the following stages:
             created
[4]
             activation started
[5]
             activation completed
             start to execute body
[6]
```

Figure 9.1 - Nested Tasks.

lems are proposed. Figure 7.3 is reproduced here as figure 9.1 for clarity. As before, task creation by allocators is not discussed. In most cases, the solutions follow the spirit of the language by copying closely the language semantics defined for situations where tasks are aborted or exceptions raised during activation. Three exceptions to this rule are:

(1) In Ada, aborts are propagated to descendents, exceptions to callers. PROCESSOR_FAILURE exceptions will go in both directions. In fact, the heartbeat mechanism will enable all other processes to know of a processor failure. What messages are sent to each process is determined by the fake message mechanism.

(2) It is suggested that, under certain circumstances, a task should be allowed to continue even though its parent is on a processor which has failed. This is the case when P fails at [2].

(3) P waits at its BEGIN until A and B have completed their activations. When A is aborted at [4] or [5], P will wait for ever. This is assumed to be an oversight in the language definition. When A fails at [4] or [5] a PROCESSOR_FAILURE exception will be sent to P. P will thus have a chance to handle the exception, by starting a similar task on another processor, for example.

What should be done when a processor fails during definition and activation of subtasks? We discuss various cases below. The numbers refer to the numbers in figure 9.1:

P fails at [1]

A and B are created and can accept entry calls. The callers are then trapped, since A and B will never be activated. When knowledge of P's failure reaches P1 and P2 (the processors A and B are running on), A and B should be terminated. This will have the effect of raising TASKING_ERROR in any callers. In fact, in this example there cannot be any callers, since the body of a task initiated by an allocator would have to be in an outer block. The point is that P1 and P2 will have no way of knowing whether P failed at [1] or [2]; at [2] A could call B.

P fails at [2]

A can continue with its activation; B should be terminated. There is now a fundamental decision to be made. Should A be allowed to continue even though its parent is no longer there. On the one hand this runs against the grain of the language, on the other A will get into no trouble until it tries to communicate with P, then TASKING_ERROR will be raised. The problem of global variables can be dealt with by treating them as local variables with implicit updates at synchronization points.

P fails at [3]

Similar to the previous case except that A and B can both continue.

A fails at [4] or at [5]

A PROCESSOR_FAILURE exception should be raised in P. If this is not done P will be trapped, waiting for the completion of A's activation.

A fails at [6]

This will have no direct effect on either P or B unless they try to communicate with A.

9.4. RECONFIGURATION

Reconfiguration is controlled by a task RECONFIGURATION which runs on each processor. When informed of a failure this task starts up the replacement tasks that will run on that particular processor.

Reconfiguration can depend on many factors, the state of the computation, number of processors remaining, which tasks were on the failed processor. The task shown in figure 9.2 bases the reconfiguration solely on which processor failed; it could easily be extended to include other factors.

9.5. STARTING SUBSTITUTE TASKS

When a task needs to be started because of a processor failure it will often be the case that speed of starting is important. Rather than have a task begin its activation when the processor failure becomes known, it is possible to activate the task initially and have it remain dormant until needed. While this requires space for the dormant tasks, it does not require processor time once the activations have been completed, since the dormant tasks will remain suspended, waiting at an accept statement until they are needed.

The task X shown in figure 9.3 would be started normally by a call X.NORMAL_START. If X had to be started because of a processor failure it would be started by a call X.START(D). In this case X would be supplied

```
task RECONFIGURATION is
   entry FAILURE( P :in PROCESSOR_ID);
end RECONFIGURATION;

task body RECONFIGURATION is
   -- body for processor i
begin
   loop
      select
         accept FAILURE( P :in PROCESSOR_ID) do
            case P is
               when P1 => -- code to reconfigure processor Pi
                          -- when processor P1 fails
                  .
                  .
               when Pn => -- code to reconfigure processor Pi
                          -- when processor Pn fails
            end case;
         end FAILURE;
      or
         terminate;
      end select;
   end loop;
end RECONFIGURATION;
```

Figure 9.2 Definition of Task RECONFIGURATION

with a consistent set of data D.

9.6. GETTING CONSISTENT DATA

To ensure a consistent data base a task DATA_CONTROL runs on each processor. DATA_CONTROL accepts data from its local tasks and sends it to all other processors. DATA_CONTROL also accepts data from other processors and uses a two-phase commit to keep a set of data consistent with the other processors. When tasks are restarted it is this set of data that is used.

```
task X is
   entry START(D:in DATA);
   entry NORMAL_START;
   -- any other entries
end X;

task body X is
   -- declarations

begin
   select
      accept NORMAL_START;
   or
      accept START(D:in DATA) do
      -- use data to initialize variables
      end START;
   end select;
   -- 'normal' begin-end part  of X follows
end X;
```

Figure 9.3 - Definition of Task X

```
task DATA_CONTROL is
   entry LOCAL_DATA_IN(TID : in TASK_ID;D : in DATA);
   entry DATA_IN( TID : in TASK_ID;D:DATA;OK:out boolean);
   entry COMMIT(P:PROCESSOR_ID;DID:DATA_ID);
   entry FAILURE(P:PROCESSOR_ID);
   entry GET_CONSISTENT_DATA(TID:in TASK_ID; D: out DATA);
end DATA_CONTROL;

task body DATA_CONTROL is
    -- declarative part omitted
begin
  loop
    select
     accept LOCAL_DATA_IN( TID: in TASK_ID; D : in DATA) do
        -- accept data from local task
        -- send it to DATA_CONTROL in all other processors
        -- when OK = TRUE received from all processors
        -- send COMMIT  to all other processors
     end LOCAL_DATA_IN;
   or
     accept DATA_IN( TID : in TASK_ID;D:DATA;OK:out boolean) do
        -- accept data from DATA_CONTROL on another processor
        -- data is stored but not committed
     end DATA_IN;
   or
     accept COMMIT(P:PROCESSOR_ID;DID:DATA_ID) do
        -- commit data with given ID to consistent data storage
     end COMMIT;
  or
     accept FAILURE(P:PROCESSOR_ID) do
        -- no longer wait for commits from P
        -- no longer send data to P
     end FAILURE;
  or
    accept GET_CONSISTENT_DATA(TID:in TASK_ID; D: out DATA) do
      -- supply consistent data so that a task can be restarted
    end GET_CONSISTENT_DATA;
  or
    terminate;
 end select;
end loop;
end DATA_CONTROL;
```

Figure 9.4 - Definition of Task DATA_CONTROL

9.7. OTHER RESTRICTIONS

For the scheme outlined above to work, some restrictions must be placed on the user's programs. Firstly, there should be no global variables. It is extremely difficult (perhaps impossible) to keep them consistent, for, while the latest value is being distributed to other processors, a local task may update the variable. Secondly, nested tasks should either not be allowed at all, or else restricted, by requiring that if a unit contains nested tasks, that unit and all the tasks nested within it should run on the same processor.

Given these restrictions, the heartbeat mechanism, the fake message system, the new exception, and the various tasks discussed in this section, we feel that Ada programs can be built to survive processor failure in a distributed system.

10. CONCLUSION

Our main conclusion is that Ada is not well suited to distributed processing where reliability is a concern. It is our opinion that any language which is to be used on a distributed system must face the fact that processors may fail. A distributed system which cannot cope with loss of a processor is no better than a uniprocessor system, and the expense and overhead incurred in distribution is wasted. Any language which does not provide tools to facilitate recovery is not really suitable for programming a distributed system.

The specific difficulties in the Ada language which we discussed in Section 7 are very serious though not overwhelming. We note that these difficulties are not unique to Ada. No language proposal has been made which is entirely satisfactory for programming an unreliable distributed system. This is a very difficult problem but the fact that Ada essentially ignores it is surprising and depressing.

The fact that the Ada rendezvous semantics are so poorly defined that we cannot decide what the language means is really unforgivable. We have discussed these points with members of the Ada language design team (R. K. B. Dewar and P. F. Hilfinger) and have not received an explanation which we find acceptable. Many of our concerns were presented to the Ada community in writing at the AdaTEC meeting in June 1982 to no avail. The very poor state of definition of the language is underscored by the fact that substantial changes in the language occurred even between the July 1982 Ada definition and the ANSI Ada definition. For example, the mechanism for updating shared variables

has been completely redefined.

The proposals we have made for error detection (software heartbeats) are adequate for most expected situation but we have no good estimate of the overhead penalty they will introduce. It can be argued that overhead is irrelevant since error detection is essential. Studies of the most effective means of implementing heartbeats seem worthwhile. It might be appropriate to consider using hardware assistance to generate and monitor heartbeats.

Our techniques for preventing hardware failure during rendezvous from suspending unsuspecting tasks indefinitely seem satisfactory. Once again we have no real estimates of the likely overhead involved in using this system of "fake" messages.

The programming techniques we propose to allow for redundant tasks to be available on separate processors are quite general though they do impose a substantial structure on programs. The further limitations on global variables and the rules for assignment of nested tasks to processors impose an even more rigid structure. We feel there is no alternative if reconfiguration following failure is to be possible. At this point we feel confident that Ada programs can be written for unreliable distributed systems, and that the results will provide satisfactory reconfiguration. The proof of these ideas will be the construction of a demonstration system which we are presently pursuing. No doubt many important issues will be raised as we undertake this implementation.

An outcome of the work described in this report is the use of the model of distributed processing discussed in Section 6 and the Ada

difficulties discussed in Section 7 to suggest better language structures and possible changes to Ada. We have already considered a range of changes to Ada but so far none have proved to be very useful. We will continue this work. Another possibility is experimentation with new language structures in the framework of a new language for distributed processing.

REFERENCES

(1) Programming Manual For The Ada Programming Language, U. S. Department of Defense, July 1982.

(2) Personal Communication, CODENOLL Corporation, New York, 1982.

(3) Hansen, P. M., et al, "A Performance Evaluation Of The iAPX 432" Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California, May, 1982.

(4) Wensley, J. H. et al, "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

(5) Hopkins, A. L., et al, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

(6) Formal Definition Of The Ada Programming Language, Cii Honeywell Bull, November 1980.

(7) Pratt, T. W., "H-graph Semantics", Technical Report Numbers 81-15, 81-16, University of Virginia, 1981.

(8) Tanenbaum, A. S., "Network Protocols", ACM Computing Surveys, Vol. 13, No. 4, December 1981.

Appendix 1

An operational model of Ada tasking has been devoloped using an H_graph notation developed in, 'H_Graph Semantics', T.W.Pratt, Technical Report Department of Applied Mathematics and Computer Science, University of Virginia,Sept 1981.

In the model each task is assummed to be running on a separate processor: communication between processes (and hence between processors) is by remote calls to kernel procedures. The run time state is described by an H_graph grammar (A). Every instruction will ultimately be defined as a transformation of the run time state.

The transform COMPILE (B) translates an Ada text into an intermediate form consisting of a list of declarations and a list of executable statements (ie transformations of the run time state). Execution of the intermediate form proceeds in two steps. First elaboration of the declaration list, and second, execution of the statement list.

An example of Ada text (C) and the intermediate form obtained by the application of COMPILE (D) is included.

A.

```
network ::=
    [ [NETWORK]
        -<node_id>->   network_node
       {-<node_id>->   network_node }
    ]

network_node ::=
    [ [NETWORK_NODE]
        -name-> [ <node_id> ]
        -processor->   processor
        -communications_interface-> [ [COMM]
                                                -proc_export->   proc_info
                                  -proc_import_queue-> *piq: QUEUE( proc_info )
                                ]
       -kernel_proc_code-> [ [KPCODE]
                                -<kproc_name>->   code
                               {-<kproc_name>->   code }
                           ]
       -process->   process
    ]

processor ::=
    [   [PROCESSOR]
        -next_instruction->   instruction_pointer
        -timer-> [ [TIMER_HARDWARE ]
```

```
                        -set-> [ <timer_status> ]
                        -delay-> [ <time> ]
                        -transfer_address->   instruction_pointer
                    ]
        -flags-> [ [FLAGS]
                        -user_pgm_suspended-> [ <boolean> ]
                        -inhibit_timer-> [ <boolean> ]
                -inhibit_abort-> [ <boolean> ]
                -inhibit_exception-> [ <boolean> ]
                -check_immediate_rendezvous-> [ <boolean> ]
            ]

        -network_node-> [ network_node ]
        -kproc_info-> proc_info
    ]

instruction_pointer ::= [ [IP]
                            -instruction-> [ code_node ]
                            -code_block->   [ code ]
                        ]

proc_info ::=
    [   [ PROC_INFO ]
        -to-> [ <node_id> ]
        -from-> [ <node_id> ]
        -kproc_name-> [ <name> ]
```

```
        -parameters-> KEYED_LIST( <integer>,arb_node )
    ]

code ::= LIST( code_node )

code_node ::= instruction_node | branch_node | LIST( code_node )

instruction_node ::= [ [INSTRUCTION_NODE]
                        -transform-> [ <transform_id> ]
                        -arguments-> KEYED_LIST( <integer>,arb_node )
                     ]

branch_node ::= [ [BRANCH_NODE]
                    -condition-> function_node
                    -alternatives-> KEYED_LIST( <integer>,code )
                  ]

function_node ::= [ [FUNCTION_NODE]
                      -function-> [ <function_id> ]
                      -arguments-> KEYED_LIST( <integer>,[ arb_atom ]
                      -result-> [ arb_atom ]
                    ]

process ::= [ [PROCESS]
              -process_object->   process_object
```

```
                -proc_import_queue-> **piq
                ]

process_object ::= [ [PROCESS_OBJECT]
                    -next_instruction->   instruction_pointer
                    -activation_record_stack-> STACK( activation_record )
                    -load_module-> [ load_module ]
                ];

activation_record ::= task_activation_record | subprogram_activation_record | package activation record


task_activation_record ::=
    [ [TAR]
        -user_data->   user_data
        -system_data->   system_data
        -elaboration_data-> elaboration_data
    ]

user_data ::=
                [ [USER_DATA]
                    -locals-> --allocated by elaboration
                    -non_locals-> KEYED_LIST( <name>,[<nesting_level>] )
                ]

system_data ::=
    [   [SYSTEM DATA]
```

```
    -my_phone_#-> processing_unit
    -state-> [ <state> ]
    -context->      [ [CONTEXT]
                    -ref_stack-> [ display ]
                    -with_list-> [              ]
                    -use_list-> [               ]
                            ]
        -exception_list-> LIST( <exception_name> )
        -governor->   processing_unit
        -dependent_task_list-> LIST( dependent_task_info )
        -#dependent_tasks_not_terminated-> [ <integer> ]
        -#noisy_tasks_in_dependent_task_tree-> [ <integer> ]
        -entry_called-> [ <entry_name> ]
        -entry_list-> KEYED_LIST( <entry_name>,entry_list_node )
        -list_of_handlers->     handlers_list_node
        -ready_to_rendezvous_list-> LIST( entry_name )
        -nesting_level-> [ <integer> ]

 ]


display ::= KEYED_LIST( <nesting_level>,processing_unit )

dependent_task_info ::=
                                [ [ DEPENDENT_TASK_INFO]
                                        -processor-> processing_unit
                                        -terminated-> [ <boolean> ]
                                -tree_quiet-> [ <boolean> ]
```

```
                    ]


entry_list_node ::=
    [   [ENTRY_LIST_NODE]
          -state-> [ <state> ]
          -transfer_address-> [ instruction_pointer ]
          -queue-> QUEUE( entry_queue_node )
    ]


entry_queue_node ::= [ [ENTRY_QUEUE_NODE]
                              -processor-> processing_unit
                              -parameters-> KEYED_LIST( <integer>,arb_node )
                         ]


handlers_list_node ::=
    [     [HANDLERS_LIST_NODE]
       -in_handler-> [ boolean ]
       -list-> KEYED_LIST( <exception_name>,instruction_pointer )

    ]


processing_unit ::=
    [ [PROCESSING_UNIT]
        -network_node-> [ <node_id> ]
        -tar->    [ task_activation_record ]
    ]
```

```
elaboration_data ::= [ [ELABORATION_DATA]
                              -task_activation_data-> task_list_node
                     -allocator_execution_data-> task_list_node
                    ]

task_list_node ::= [ [TASK_LIST_NODE]
                              -#_non_allocated_tasks-> [ <integer> ]
                     -#_non_active_tasks-> [ <integer> ]
                     -list-> KEYED_LIST( <task_name>,task_info )
                   ]

task_info ::= [ [TASK_INFO]
                        -name-> full_id
                -processor-> [ processing_unit ]
                        -allocation_completed-> [ <boolean> ]
                -activation_completed-> [ <boolean> ]
                        -load_module-> [ load_module ]
               ]

subprogram_activation_record ::=
            [ [SAR]
                -id-> full_id
              -context-> display
                -user_data-> user_data
              -body-> subprogram_body
              -return_address-> instruction_pointer
```

```
            -dependent_task_list-> LIST( dependent_task_info )
            -#_dependent_tasks_not_terminated-> [ <integer> ]
            -task_activation_data-> task_list_node
            -allocator_execution_data-> task_list_node
        ]

load_module ::=
        [ [LOAD_MODULE]
            -module_id-> full_id
            -entries-> LIST( entry_node )
            -body-> task_body
            -context_of_body-> display
            -governor-> processing_unit
            -activator-> processing_unit
        ]

entry_node ::= [ [ENTRY_NODE]
                -name-> full_ident
                        -range-> range
                -formal_params-> formal_part
            ]

range ::=  [ [RANGE]
                -low-> [ <integer> ]
                -high-> [ <integer> ]
            ]
```

```
full_id ::=      [ [FULL_IDENT]   ]
                          -id-> [ <identifier> ]
                          -level-> [ <nesting_level> ]
                    ]


body ::=                        subprogram_body
                                !task_body


subprogram_specification ::=        [ [SUBPROGRAM_SPECIFICATION]
                                  -id-> [ <identifier> ]
                                                -level-> [ curr_level ]
                                  -params-> formal_part
                                  ]


formal_part ::= LIST([parameter_specification] )


parameter_specification ::=      [ [PARAMETER_SPECIFICATION]
                                 -id_list-> identifier_list
                                                -level-> [ <integer> ]
                                 -mode-> mode
                                 -type-> type_mark
                                 -value-> code
                                ]
```

```
mode ::=   [IN] | [IN  OUT] | [OUT]

subprogram_body ::=      [ [SUBPROGRAM_BODY]
                          -specifications-> subprogram_specification
                          -declarations-> declarative_part
                          -statements-> code
                          -exceptions-> LIST( {exception_handler} )
                         ]

task_body ::= [ [TASK_BODY]
                -name-> full_id
                -declarations-> declarative_part
                -statements-> code
                -exceptions-> LIST( exception_handler )
               ]

exception_handler ::= [ [EXCEPTION_HANDLER]
                        -name_list-> LIST( exception_choice )
                        -handlers_code-> code
                        ]

exception_choice ::= [ exception_name ] | [ OTHERS ]

QUEUE(X) ::=
```

```
[ [QUEUE]
    -first-> [ QUEUE_ELEMENT( X ) ]
    -last->  [ QUEUE_ELEMENT( X ) ]
]

QUEUE_ELEMENT( X ) ::= [ # ]    |
[ [QUEUE_ELEMENT]
    -head->    [ X ]
    -rest->    [ QUEUE_ELEMENT( X ) ]
]

KEYED_LIST( <KEY>,MEMBER ) ::=
[ # ] | [ [KEYED_LIST]
          -<KEY>-> MEMBER
        [ -<KEY>-> MEMBER ]
      ]

LIST( MEMBER ) ::= [#] | [ [LIST]
                         [ --> MEMBER  ]
                           --> [#]
                       ]

STACK( KIND ) ::=  [#] | [ [STACK]
                         -head-> [ KIND ]
                         -tail-> [ STACK( KIND ) ]
                       ]
```

<node_id> ::= <identifier>

<name> ::= <identifier>

<kproc_name> ::= <identifier>

<entry_name> ::= <identifier>

<transform_id> ::= <identifier>

<function_id> ::= <identifier>

<timer_status> ::= ON | OFF

<time> ::= <integer>

<boolean> ::= TRUE | FALSE

B.

```
transform [COMPILE]
            —> *ADA_TEXT: in [ <subprogram_body> ]
            —> *MAIN_BODY: out subprogram_body

var

*COMPILE_TIME_INFO: compile_time_info := 0 [#]


compile_time_info ::= [ [COMPILE_TIME_INFO] ]

                          -level-> [ <nesting_level> ]

                          -context-> context

                      ]


context ::= KEYED_LIST( { 1 .. n},name_table )


name_table ::= LIST( name_table_item )


name_table_item ::= [ [NAME_TABLE_ITEM] ]

                        -id-> full_id

                        -type_name-> type_name

                        -declaration-> [ basic_declaration ]

                    ]



full_id ::= [ [FULL_ID]  ]

                    -id-> [ <identifier> ]

                    -level-> [ <nesting_level> ]

                ]


<nesting_level> ::= <integer>


type_mark           -- see productions
basic_declaration   --  in the pair grammar
```

```
KEYED_LIST( { <KEY> },MEMBER ) ::=
              [#] |  [ [KEYED_LIST]
                          -<KEY>-> MEMBER
                        { -<KEY>-> MEMBER }
                       ]

LIST( MEMBER ) ::= [#] |  [ [LIST]
                               { --> MEMBER }
                                 --> [#]
                             ]

-- curr_level represents an integer with value *COMPILE_TIME_INFO/.level'

begin
   parse
     *ADA_TEXT:[ <subprogram_body> ]
   generate
     *MAIN_BODY: subprogram_body#1
     *COMPILE_TIME_INFO: subprogram_body#2

pair grammar
basic_declaration ::=
                     object_declaration
                     | type_declaration
                     | subprogram_declaration
                     |task_declaration
                     |exception_declaration
```

```
basic_declaration ::=
                    object_declaration
                    | type_declaration
                    | subprogram_declaration
                    |task_declaration
                    |exception_declaration

                ;

object_declaration ::=
                   identifier_list : [constant] subtype_indication [:= expression]

object_declaration ::= #1 *a:[ [OBJECT_DECLARATION]
                     -id_list-> identifier_list
                     -type-> subtype_indication
                     -level-> [ curr_level ]
                     -value-> expression
                                 -allocated-> [ <boolean> ]
                    ]
                        #2 add_name_to_name_table(curr_level,identifier_list,subtype_indicati

                ;

identifier_list ::= identifier {, identifier }

identifier_list ::= LIST( s:{[ <identifier> ]} )
```

```
        s = { identifiers in RHS of LHS production }

        ;

type_declaration ::=
            type identifier [discriminant_part] is type_definition

type_declaration ::= #2 *a: type_definition

            add_name_to_name_table(curr_level,identifier,type_definition,[*a])

        ;

type_definition ::= access_type_definition

type_definition ::= access_type_definition

        ;

subtype_indication ::= type_mark [constraint]

subtype_indication ::= [ [SUBTYPE_INDICATION]
                         -type_info-> type_mark
                         -constraint-> constraint
                       ]

        ;
```

```
type_mark ::= type_name | subtype_name

type_mark ::= type_name | subtype_name

                    if type_name = pdtype then
                    type_mark ::= [ [PREDEFINED]
                                    -pdtype-> [ <pdtype> ]
                                ]
                    else
                    *tn := LOCATE( type_name )
                    type_mark ::= *tn/type_info
                    endif
                   ;


access_type_definition ::= access subtype_indication


access_type_definition ::=  [ [ACCESS_TYPE_DEFINITION]
                      -type_info-> type_mark
                      -constraint-> constraint
                              -defining_unit-> curr_unit
                  ]
                      ;


declarative_part ::= {basic_declarative_item}{later_declarative_item}
```

```
declarative_part ::= [ [DECLARATIVE_PART]
                       -basic_items-> LIST({ basic_declarative_item })
                       -later_items-> LIST({ later_declarative_item })
                     ]

                          | [ [#] ]

                     ;

basic_declarative_item ::= basic_declaration

basic_declarative_item ::= basic_declaration

                     ;

later_declarative_item ::=
                          body
                          |subprogram_declaration
                          |task_declaration

later_declarative_item ::=
                          body
                          |subprogram_declaration
                          |task_declaration

                     ;

body ::=                  subprogram_body
```

```
                        |task_body

body ::=                subprogram_body
                        |task_body

                ;


name ::=
                        simple_name
                        |indexed_component
                        |selected_component


name ::=
                        full_id
                        |indexed_component
                        |selected_component

                ;

task_simple_name_1 ::= simple_name

task_simple_name_1 ::= full_id

                        ;

task_simple_name_2 ::= simple_name
```

```
task_simple_name_2 ::= #1 full_id

                       #2 curr_level := curr_level + 1

                          add_entry_info_to_name_table(curr_level,full_id)

                                -- add entry names and parameter specifications
                                -- from the task specification to the
name_table

                          ;



simple_name ::=         identifier


full_id ::= [ [FULL_ID]

            -id-> [ <identifier> ]

            -level-> [ n ]          -- n is the level found by searching
                                    -- the surrounding contexts for the
         ]                          -- identifier.


         ;



indexed_component ::= name( expression  {, expression } )


indexed_component ::= [ [INDEXED_COMPONENT]

                 -name-> name

                 -indices-> KEYED_LIST({<integer>},expression )

                ]


                ;
```

```
selected_component ::= name.selector

selected_component ::= [ [SELECTED_COMPONENT]
                         -name-> name
                         -selector-> selector
                       ]
                       ;

selector ::=           simple_name
                       |all

selector ::=           full_id
                     | [ALL]
                     ;

allocator ::= new type_mark

allocator ::= [ [ALLOCATOR] -->
                [ REF( type_mark ) ] &t -->
                [ ALLOC( &t ) ] &ptr -->
                     [#]
              ]
              ;
```

```
sequence_of_statements ::= statement { statemant }

sequence_of_statements ::= LIST( { statement } )

                ;

statement ::=           simple_statement
                        |compound_statement

statement ::=           simple_statement
                        |compound_statement

                ;

simple_statement ::= null_statement
                     |assignment_statement
                     |delay_statement
                     |raise_statement
                     |procedure_call_statement
                     |return_statement
                     |entry_call_statement
                     |abort_statement

simple_statement ::= null_statement
                     |assignment_statement
                     |delay_statement
```

```
                    |raise_statement
                    |procedure_call_statement
                    |return_statement
                    |entry_call_statement
                    |abort_statement

                  ;


compound_statement ::=      accept_statement
                           |select_statement


compound_statement ::=      accept_statement
                           |select_statement

                ;


null_statement ::= null;

null_statement ::= [ [NULL_STATEMENT] -->
                   [ NOOP ] -->
                   [#]
                 ]

                ;
```

```
assignment_statement ::= variable_name := expression;

assignment_statement ::= [ [ASSIGNMENT_STATEMENT] -->
                           [ REF( variable_name ) ] &z -->
                           expression  &e -->
                           [ ASSIGN( &z,&e ) ] -->
                           [#]
                         ]

                         ;

return_statement ::= return [expression];

return_statement ::= [ [RETURN_STATEMENT] -->
                       expression &e -->
                       [ RETURN( &e ) ] -->
                       --> [#]
                     ]

                       ;

subprogram_declaration ::=   subprogram_specification;

subprogram_declaration ::=   subprogram_specification;

                       ;
```

```
subprogram_specification_1 ::= procedure identifier [formal_part]

subprogram_specification_1 ::= #1 *a:[ [SUBPROGRAM_SPECIFICATION]
                          -id-> [ <identifier> ]
                          -level-> [ curr_level ]
                          -params-> formal_part
                          ]

                       #2 add_name_to_name_list(curr_level,identifier,subprogram,[*a])

                    ;

subprogram_specification_2 ::= #1 [ [SUBPROGRAM_SPECIFICATION]
                          -id-> [ <identifier> ]
                                    -level-> [ curr_level ]
                          -params-> formal_part
                          ]

                        #2 curr_level := curr_level + 1

                                ;

formal_part ::= (parameter_specification {; parameter_specification}

formal_part ::= LIST({parameter_specification} )

                ;

parameter_specification ::= identifier_list : mode type_mark [:= expression]
```

```
parameter_specification ::= #1 *a:[ [PARAMETER_SPECIFICATION]
                            -id_list-> identifier_list
                            -level-> [curr_level + 1]
                            -mode-> mode
                            -type-> type_mark
                            -value-> expression
                          ]
                      #2 add_name_to_name_list(curr_level+1,identifier_list,[*a])
                      ;

mode ::= [in] | in out | out

mode ::= [IN] | [IN OUT] | [OUT]
                      ;

subprogram_body ::=
            subprogram_specification_2 is
             [declarative_part]
             begin
                sequence_of_statements
               [exception
                    exception_handler
                    {exception_handler}  ]
             end [designator];
```

```
subprogram_body ::= #1 [ [SUBPROGRAM_BODY]
                    -specifications-> subprogram_specifications
                    -declarations-> declarative_part
                    -statements-> [ prelude -->
                                    overture -->
                                    sequence_of_statements -->
                                    epilog -->
                                    [#]
                                     ]
                    -exceptions-> LIST( {exception_handler} )
                  ]
            #2 curr_level := curr_level - 1

                ;
```

```
procedure_call_statement ::= procedure_name [ actual_parameter_part];

procedure_call_statement ::= [ [PROCEDURE_CALL_STATEMENT]
                         actual_parameter_part &params -->
                         [ REF(procedure_name ) ] &p -->
                         [ CALL( &p,&params ) ] -->
                         [#]
                        ]
```

```
                    ;

actual_parameter_part ::= (parameter_association {, parameter_association})

actual_parameter_part ::= LIST( parameter_association )

                    ;

parameter_association ::= [formal_parameter =>] actual_parameter

parameter_association ::= [ [PARAMETER_ASSOCIATION] -->
                    actual_parameter -->
                    [add_to_parameter_list(param_info,param_id,mode,type)] -->     -- param_i
filled in from
            -- corresponding formal parameter
                    [#]
                ]

                ;

formal_parameter ::= parameter_simple_name

formal_parameter ::= parameter_simple_name

                ;

actual_parameter ::= expression
                    |variable_name
```

```
                    |type_mark (variable_name)

actual_parameter ::= [ [VAL] -->
                      expression &e -->
                           [fill_in_param_info(&e,'VALUE',null)] -->
                      [#]
                    ]

                    | [ [REF] -->
                     [ REF( variable_name ) ] &n  -->
                              [fill_in_param_info(&n,'ADDR',null)] -->
                     [#]
                      ]

                    | [ [REFT] -->
                     [REF(variable_name)] &n -->
                     [REF(type_mark) ] &t -->
                              [fill_in_param_info(&n,'TADDR',&t)] -->
                              [#]
                      ]


                        ;


task_declaration ::= task_specification;

task_declaration ::= task_specification;
```

```
;

task_specification ::=
        task [type] identifier [is
           {entry_declaration}
           {representation_clause}
        end [task_simple_name]]


load_module_template ::= #2 *a:[ [LOAD_MODULE_TEMPLATE]
                                  -module_id-> [ [FULL_ID]
                                      -id-> [ <identifier> ]
                                      -level-> [ curr_level ]
                                                ]
                        -entries-> LIST( {entry_declaration} )
                                  -body-> [#]
                                  -context_of_body-> [#]
                                  -governor-> [#]
                                  -activator-> [#]
                        ]

                    add_name_to_name_list(curr_level,identifier,task,[*a])


                    ;

task_body ::=
      task body task_simple_name_2 is
```

```
        [ declarative_part ]
      begin
           sequence_of_statements
      [exception
           exception_handler
           {exception_handler}
      end [task_simple_name];


task_body ::= #2 *body: [ [TASK_BODY]
                        -name-> task_simple_name_2
                        -declarations-> declarative_part
                        -statements-> [prologue -->
                                       overture -->
                                  sequence_of_statements -->
                                  epilog -->
                                  [#] ]
                        -exceptions-> LIST( exception_handler )
                       ]

           find_in_name_list( task_simple_name_2,*n )
           *n/body' := task_body'
        curr_level := curr_level - 1


                       ;


entry_declaration ::=
```

```
   entry identifier [(discrete_range)] [formal_part];


entry_declaration ::= #1 [ [ENTRY_DECLARATION]
                      -id-> identifier
                      -level-> [ curr_level ]
                      -range-> range
                      -formal_part-> formal_part
                     ]

                   #2 add_name_to_name_list(curr_level,identifier,entry )

                   ;


entry_call_statement_1 ::= entry_name[actual_parameter_part];


entry_call_statement_1 ::= [ [ENTRY_CALL] -->
                       [REF(entry_name) ]&E,&pssr -->
                       actual_param_part &params -->
                       [ entry_call_proc( &pssr,&E,&params ) ] -->
                       [#]
                     ]

                   ;


entry_call_statement_2 ::= entry_name[actual_parameter_part];
```

```
entry_call_statement_2 ::= [ [ENTRY_CALL] -->
                    [REF(entry_name) ]&E,&pssr -->
                    actual_param_part &params -->
                    [#]
                  ]
                ;


accept_statement_1 ::=

    accept entry_simple_name [(expression)] [formal_part] [ do
         sequence_of_statements
    end [entry_simple_name] ];

accept_statement_1::=  [ [ACCEPT_STATEMENT] -->
                [REF(entry_simple_name) ] &n -->
                expression &e -->
                [REF(formal_part)] &f -->
                [ accept_proc(&e,&f,&n) ] -->
                sequence_of_statements -->
                [ end_of_rendezvous ] -->
                [#]
               ]


               ;
```

```
accept_statement_2 ::= accept_part_1 accept_part_2

accept_part_1 ::= entry_simple_name [ (expression) ] [formal_part]

accept_part_2 ::= [ do sequence_of_statements end [entry_simple_name] ];

accept_statement_2 ::= accept_part_1 accept_part_2

accept_part_1::=     [ [ACCEPT_PART_1] -->
                [REF(entry_simple_name) ] &n -->
                expression &e -->
                formal_part &f -->
                        [#]
                    ]


accept_part_2::=    [ [ACCEPT_PART_2] -->
                [ accept_proc(&e,&f,&n) ] -->
                sequence_of_statements -->
                [ end_of_rendezvous ] -->
                [#]
                   ]

                   ;


delay_statement_1 ::= delay simple_expression;

delay_statement_1 ::= [ [DELAY_STATEMENT_1] -->
               simple_expression &d -->
               [ set_timer ( &d,*a ) ] -->
               [ state_becomes( ' suspended:at delay ' ) ] -->
```

```
                *a:[#]
                 ]


                 ;

delay_statement_2 ::= delay simple_expression;

delay_statement_2 ::= [ [DELAY_STATEMENT_2] -->
                simple_expression &d -->
                [#]
                 ]


                 ;

select_statement ::= selective_wait
                    |conditional_entry_call
                    |timed_entry_call


select_statement ::= selective_wait
                    |conditional_entry_call
                    |timed_entry_call


                      ;

selective_wait ::=
      select
```

```
        select_alternative
        {or
          select_alternative}
        else
           sequence_of_statements
    end select;

selective_wait ::= [ [SELECTIVE_WAIT_STATEMENT] —›
                     [ set_up_temp_data_str ] —›
                     LIST({ select_alternative}) —›
                     [ [BRANCH]
                -condition-› [check_if_any_open_guard]
                -alternatives-› [ [KEYED_LIST]
                              -true-› [ perform_select ]
                              -false-› sequence_of_statements
                           ]
               ] —›
               [ release_temp_data_str ] —›
               **end_of_select:[#]
        ]

        ;

selective_wait ::=
      select
          select_alternative
```

```
        {or
          select_alternative}
      end select;

selective_wait ::= [ [SELECTIVE_WAIT_STATEMENT] -->
                     [ set_up_temp_data_str ] -->
                     LIST({ select_alternative}) -->
                     [ [BRANCH]
                  -condition-> [check_if_any_open_guard]
                  -alternatives-> [ [KEYED_LIST]
                                -true-> [ perform_select ]
                                -false-> [ RAISE_EXCEPTION('SELECT ERROR' ) ]
                              ]
                ] -->
                [ release_temp_data_str ] -->
                **end_of_select:[#]
        ]

        ;

select_alternative  ::= [ when condition => ]
                            selective_wait_alternative

select_alternative ::= [ [SELECT_ALTERNATIVE] -->
                  [ [BRANCH]
                      -condition-> condition
                      -alternatives-> [ [KEYED_LIST]
```

```
                                -true-> selective_wait_alternative
                                -false-> [#]
                                 ]
                        ] -->
                                [#]
                        ]

                        ;


selective_wait_alternative ::= accept_alternative
                               |delay_alternative
                               |terminate_alternative


selective_wait_alternative ::= accept_alternative
                               |delay_alternative
                               |terminate_alternative


                           ;


accept_alternative ::= accept_statement_2 [sequence_of_statements]

accept_statement_2 ::= accept_part_1 accept_part_2


accept_alternative ::= [ [ACCEPT_ALTERNATIVE] -->
                         accept_ part_1 &n &e &f -->
                                 *a
```

```
                    *b: accept_part_2 -->
                sequence_of_statements -->
                **end_of_select
                *a:[put_on_open_guards_list( &n,&e,&f,*b ) -->
                [#]
               ]

               ;

delay_alternative_1 ::= delay_statement_2 [sequence_of_statements]

delay_alternative_1 ::= [ [DELAY_ALTERNATIVE] -->
               delay_statement_2 &d -->
                        *a
               *c:sequence_of_statements -->
               **end_of_select
               *a:[ update_smallest_open_delay( &d,*c ) ] -->
               [#]
                ]

              ;

terminate_alternative ::= terminate

terminate_alternative ::= [ [TERMINATE_ALTERNATIVE] -->
```

```
                ;

timed_entry_call ::=
        select
           entry_call_statement
           [sequence_of_statements_1]
        or
            delay_statement_2
            [ sequence_of_statements_2 ]
        end select;

timed_entry_call ::= [ [TIMED_ENTRY_CALL] -->
                  delay_statement_2 &d  -->
                        *b
                        *a:sequence_of_statements_2 -->
                  *end
                        *b:[ set_timer( &d,&a ) ] -->
                  entry_call_statement_1 -->
                  sequence_of_statements_1 -->
                   *end:[#]
                ]

                ;

abort_statement ::= abort task_name {,task_name};
```

```
                    [ set_open_terminate_flag ] -->
                    [#]
                  ]

                  ;

conditional_entry_call ::=
        select
          entry_call_statement_2
          [sequence_of_statements_1]
         else
          sequence_of_statements_2
         end select;

conditional_entry_call ::= [ [CONDITIONAL_ENTRY_CALL] -->
                    entry_call_statement_2  &E,&pssr,&params -->
                    [ request_rendezvous (&E,&pssr,$params) ] -->
                    [ [BRANCH]
                    -condition-> [ rendezvous_possible( ) ] -->
                    -alternatives-> [ [KEYED_LIST]
                                -true-> sequence_of_statements_1
                                -false-> sequence_of_statements_2
                              ]
                    ] -->
                    [#]
                  ]
```

```
abort_statement ::= LIST( (ABORT_TASK(task_name) })

ABORT_TASK(X) ::= [ [ABORT] —>
                    [REF(X) ] &n —>
                    [ abort_exec( &n ) ] —>
                    [#]
                  ]

                  ;
```

```
exception_declaration ::= identifier_list : exception;

exception_declaration ::= [ [EXCEPTION_DECLARATION]
                            -id_list-> identifier_list
                            -type-> [EXCEPTION]
                          ]

                          ;
```

```
exception_handler ::=
   when exception_choice {|exception_choice } =>
        sequence_of_statements

exception_handler ::= [ [EXCEPTION_HANDLER]
                        -name_list-> LIST( exception_choice )
                        -handlers_code-> sequence_of_statements
```

```
                ]

                ;


exception_choice ::= exception_name

                    |others

exception_choice ::= [ exception_name ] | [ OTHERS ]

                ;


raise_statement ::= raise [exception_name];

raise_statement ::= [ [RAISE_STATEMENT] -->
                [REF( exception_name ) &n -->
                [ raise_exception ( &n ) ] -->
                [#]
               ]

end COMPILE
```

C.

```
procedure  FIRST is
  task type SIMPLE is
    entry X( I: in integer );
  end SIMPLE;

  task type COMPUTE is
    entry Y ( I: out integer );
  end COMPUTE;

  S:SIMPLE;
  C:COMPUTE;
  I:integer :=6;

  task body SIMPLE is
    A:integer :=10;
    B:integer;
    begin
      accept X (I:in integer);
         B := A + I;
      end X;
      print(B);
    end SIMPLE;

  task body COMPUTE is
    C:integer :=3;
    begin
      S.X(C);
```

```
    accept Y(I:out integer);
      I := C + 2;
    end Y;
  end COMPUTE;

begin  -- FIRST
  C.Y(I);
  I := I + 5;
  PRINT(I);
end FIRST;
```

D.

```
[ [SUBPROGRAM_BODY]
   -specifications-> [ [SUBPROGRAM_SPECIFICATION]
                        -id-> [FIRST]
                        -level-> [0]
                        -params-> [#]
                     ]

   -declarations-> [ [LIST] -->
               [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] --> [S] --> [#] ]
                  -type-> [ [SUBTYPE_INDICATION]
                                 -type_info-> *lmtSIMPLE
                                 -constraint-> [#]
                       ]
                  -level-> [1]
                  -value-> [#]
               ] -->

               [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] --> [C] --> [#] ]
                  -type-> [ [SUBTYPE_INDICATION]
                                 -type_info-> *ltmCOMPUTE
                                 -constraint-> [#]
                       ]
                  -level-> [1]
                  -value-> [#]
```

```
        ] —>

              [ [OBJECT_DECLARATION]
            -id_list-> [ [LIST] —> [I] —> [#] ]
            -type-> [ [PREDEFINED]
                                -pdtype-> [INTEGER]
                            ]
            -level-> [1]
            -value-> [#]
        ] —>

           [#]
    ]

-statements-> [ [LIST] —>
        prologue —>
        overture —>
              [ [ENTRY_CALL] —>
          [ REP( C,1),(Y,1)]  &pssr,&E —>
          [REP( I,1 ) ] &params —>
          [entry_call_proc(&pssr,&E,&params)] —>
          [#]
              ] —>

        [ [ASSIGNMENT_STATEMENT] —>
          [ REP( (I,1) ) ] &addr —>
          [ [EXPRESSION] —>
```

```
                [REF( (I,1) ) ] &a -->
                [REF( 5 )     ] &b -->
                [ADD( &a,&b )] &c -->
                [#]
              ] -->
              [ ASSIGN( &addr,&c ) ] -->
              [#]
            ] -->
            [REF( I,1 ) ] &out -->
            [PRINT(&out) ] -->
            epilog -->
            [#]
            ]

   -exceptions-> [#]
]

*lmtSIMPLE:          [ [LOAD_MODULE_TEMPLATE]
                        -module_id-> [ [FULL_ID ]
                                       -id-> [SIMPLE]
                                       -level-> [1]
                                     ]
                      -entries-> [ [LIST] -->
                                   [ [ENTRY_DECLARATION]
                                   -id-> [X]
                                   -level-> [2]
```

```
                    -range-> [#]
                    -formal_part-> [ [LIST] -->
                              [ [PARAMETER_SPECIFICATION]
                                 -id_list-> [ [LIST] --> [I] --> [#] ]
                                 -level-> [2]
                                 -mode-> [IN]
                                 -type-> [ [PREDEFINED]
                                                          -pdtype-> [INTEGER]
                                                       ]

                                 -value->[#]
                              ] -->
                              [#]
                               ]
                         ] -->
                         [#]
                          -body-> *task_body_simple
                          -context_of_body-> [#]
                          -governor-> [#]
                          -activator-> [#]
                    ]

*lmtCOMPUTE:        [ [LOAD_MODULE_TEMPLATE]
                       -module_id-> [ [FULL_ID ]
                                      -id-> [COMPUTE]
                                      -level-> [1]
                                    ]
```

```
-entries-> [ [LIST] —>
           [ [ENTRY_DECLARATION]
           -id-> [Y]
           -level-> [2]
           -range-> [#]
           -formal_part-> [ [LIST] —>
                     [ [PARAMETER_SPECIFICATION]
                        -id_list-> [ [LIST] —> [I] —> [#] ]
                        -level-> [2]
                        -mode-> [OUT]
                        -type-> [ [PREDEFINED]
                                                    -pdtype-> [INTEGER]
                                                 ]
                        -value->[#]
                     ] —>
                     [#]
                      ]
          ] —>
          [#]
           -body-> *task_body_compute
           -context_of_body-> [#]
           -governor-> [#]
           -activator-> [#]
      ]

*task_body_simple:[ [TASK_BODY]
```

```
 -name-> [ [FULL_ID]
-id-> [S]
-level-> [2]
]
 -declarations-> [ [LIST] —>
                    [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] —> [A] —> [#] ]
                  -type-> [integer]
                  -level-> [2]
                  -value-> [10]
                ] —>

                   [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] —> [B] —> [#] ]
                  -type-> [ [PREDEFINED]
                                  -pdtype-> [INTEGER]
                              ]

                  -level-> [2]
                  -value-> [#]
                ] —>
                    [#]
              ]

-statements-> [  [LIST] —>
          prologue—>
          overture —>
```

```
                [ [ACCEPT_STATEMENT] -->
               [REP( X,1 ) ] &n -->
               [REP( I,2 ) ] &f -->
               [accept_proc(&f,&n)] -->
                  [ [ASSIGNMENT_STATEMENT] -->
                  [REP( B,2 ) ] &addr -->
                  [ [EXPRESSION] -->
                    [REP( A,2 ) ] &op1 -->
                    [REP( I,2 ) ] &op2 -->
                    [ADD( op1,op2 ) ] &e -->
                    [#]
                                  ] -->
                     [ASSIGN( &addr,&e ) ] -->
                     [#]
                  ] -->
               [end_of_rendezvous] -->
               [#]
                 ] -->
                             [REP ( B,2 ) ] &out -->
                 [PRINT( &out ) ] -->
                 [#]
          ]
     -exceptions->[#]
]
```

```
*task_body_compute:[ [TASK_BODY]
                     -name-> [ [FULL_ID]
                    -id-> [C]
                    -level-> [2]
                    ]
                     -declarations-> [ [LIST] -->
                                     [ [OBJECT_DECLARATION]
                                   -id_list-> [ [LIST] --> [C] --> [#] ]
                                   -type-> [ [PREDEFINED]
                                                    -pdtype-> [INTEGER]
                                                 ]
                                   -level-> [2]
                                   -value-> [3]
                                ] -->
                                     [#]
                                ]
                -statements-> [ [LIST] -->
                           prologue-->
                           overture -->
                                       [ [ENTRY_CALL] -->
                          [REF( S,1),(X,1)]  &pssr,&E -->
                          [REF( C,2 ) ]&params -->
                          [entry_call_proc(&pssr,&E,&params)] -->
                          [#]
                                       ] -->
```

```
        [ [ACCEPT_STATEMENT] -->
       [REP( Y,1 ) ] &n -->
       [REP( I,2 ) ] &f -->
       [accept_proc(&f,&n)] -->
          [ [ASSIGNMENT_STATEMENT] -->
         [REP( I,2 ) ] &addr -->
         [ [EXPRESSION] -->
           [REP( C,2 ) ] &op1 -->
           [REP( 2 ) ] &op2 -->
           [ ADD( op1,op2 ) ] &e -->
           [#]
                          ] -->
         [ASSIGN( &addr,&e ) ] -->
         [#]
         ] -->
       [end_of_rendezvous] -->
       [#]
         ] -->
         epilog -->
         [#]
     ]

  -exceptions-> [#]

]
```

Appendix 2

Questions on the Ada '82 Language Reference Manual.

Questions and comments about wrong, confusing, unclear, and incomplete parts in the tasking sections (mainly chapter 9 but some of chapter 11) of intra-canvass Ada Reference Manual.

(1) General
The examples appearing in the 1982 reference manual are no more than those in the 1980 version. Most are examples of proper statement syntax only. There are places in the manual where even a simple example would clarify more than the volume of text. The reference manual needs more examples; at least one for each language feature. Where an example refers to or uses a previous example, an explicit reference should be given.

(2) Chapter 9, page 1, paragraph 1, line 1
It is stated that the execution of a program which contains no task procedes according to the rules described by the manual less chapter 9. In a multiprogrammed system, main programs look remarkably like tasks executing independently. Is the main program a task (with no entries) or not?

(3) Chapter 9, page 1, paragraph 1, line 4
"The effect of ... a program is defined in terms of a sequential execution of its actions in some order..." What does "some order" mean? We realize that the intended order is that traditionally found in implementations of Algol-descended languages with rearrangements and optimizations restricted as in chapter 11. However, the manual does not specify that, it says "some order."

(4) Chapter 9, page 1, paragraph 2, line 4
We know that two tasks are synchronized at the beginning and end of a rendezvous, and that tasks are synchronized with their declaring parent at their activation, but are there other places? For instance, are tasks "synchronized" during execution of an ABORT statement since in that case they are not operating independently. This is the first occurrence of the term "synchronize." It is a technical term in the definition of Ada semantics. It must be precisely defined.

(5) Chapter 9, page 1, paragraph 4, line 3
There are three kinds of program units of which programs can be composed according to chapter 9 but four according to chapter 7.

(6) Chapter 9, page 1, paragraph 4, line 3
What is the intent of a program unit (the term is not defined)? If I write a program unit which consists solely of a task unit or generic unit, what can I do with it? (Dare we ask "What is a main program?")

(7) Chapter 9, page 2 section 9.1 paragraph 3
"task [type] identifier [is ... end [simple_name]]" What is the distinction that is being made between an identifier and a simple_name? Is the reference manual alluding to the symbol table operation and to the relationship between the lexical analyzer and parser of a particular compiler?

(8) Chapter 9, page 2 section 9.1 paragraph 5
"...the body can ... be used for the execution of tasks designated by objects of the ... task type." From reading descriptions elsewhere in the manual it seems to us that the term "values of objects" would be more appropriate here. The continuation of the fiction that a task object and its value are somehow different when we are told that tasks behave as constants, seems silly. It does provide consistency with the descriptions of other kinds of objects and their values but can add confusing verbiage to an already confusing chapter (besides, it contributed to this mistake in the manual itself).

(9) Chapter 9, page 3 section 9.1 paragraph 1
We had an argument about when or whether it would be legal for a task to refer to itself, especially by its type identifier. We eventually came up with several valid cases, but the point here is that the manual slips the capability in and certainly does not expand on it. The material explaining how a task type name serves as a task name is very cryptic and could do with some elaboration. A separate notation for self reference would be nice.

(10) Chapter 9, page 4 section 9.2 paragraph 1
We were under the impression (and the first note in the designated section seems to support this) that task types could be passed as generic actual parameters at instantiations of generic units, yet this section, besides the note, ignores such usage. Was it ever decided whether omission in the reference manual constituted a prohibition?

(11) Chapter 9, page 5 section 9.2 note 1
The business of modes allowed and disallowed for generic parameters whose types are task types is very confusing. This note needs elaboration or it needs to be moved to an appropriate place in the chapter on generics. Why are tasks not allowed as actual parameters corresponding to generic formal parameters with mode IN since tasks by definition are "constant"?

(12) Chapter 9, section 9.3
The manual is very explicit about when task objects declared in a declarative part get activated (not before and not after the following BEGIN), and about when task objects created via an allocator get activated. When do task objects created via an allocator in the initialization of an object of an access type in a declarative part get activated? This is important in terms of understanding what is completed and what is terminated should an exception occur during the activation of one of these tasks. We do not understand when these tasks get activated! We are also concerned about the apparent inconsistency in the fates of declared and allocated tasks which experience exceptions during their activations.

(13) Chapter 9 section 9.3
In 9.3 activation is defined to be the elaboration of the declarative part of a task body. When does a task proceed after its activation has been completed? NOTE: In July 1980 Ada, section 9.3 states, ' Each task can continue its execution as a parallel entity once its activation is

completed.' Why was this omitted?

(14) Chapter 9, page 21 section 9.11
Why is task synchronization between activator and activatee only mentioned in shared variables? We need a definition of synchronization.

(15) Chapter 9, page 7 section 9.3
References at the end of section 9.3 (and probably elsewhere) still have "?" in them. Will they be replaced?

(16) Chapter 9, page 6 section 9.3 paragraphs 1 & 4
When tasks are being activated after a begin, if the activation raises an exception the task becomes completed. On the other hand paragraph 4 states that if a task has been created by the execution of an allocator and an exception is raised during its activation then the task becomes terminated. Are the cases really different and if so why? NOTE: Ch 11 p8 s11.4.2(d) says that the task would be completed in both cases.

(17) Chapter 9, page 6 section 9.3 paragraphs 1 & 4
"other tasks are unaffected" Does this include dependents or does it refer back to 'these tasks' - the tasks being activated? This is not clear.

(18) Chapter 9, page 8 section 9.4 [paragraph 2 ... Example]
Use of "unit" vs. "task" is extremely confusing. Text should replace 'certain unit' by 'parent unit'. The meaning here can be completely missed very easily (some of us did on first reading).

(19) Chapter 9, page 7 section 9.4 paragraph following (c)
The new definition of dependency needs further explanation. We suggest adding a note explaining that because of the definition of termination and the rules about leaving subprograms and blocks, only tasks defined in a unit or contained in an inner package need be checked for termination.

(20) Chapter 9, page 8 section 9.4 Example
Example is not clear because we don't know where G.ALL was activated. We suggest that comments should be ammended to read "await termination of G.ALL if it was ever activated no matter where."

(21) Chapter 9, page 10 section 9.5 (last paragraph before the example)
This needs to be rewritten more clearly. Chapter 11 section 11.5 contains a clear explanation of the situation which could be copied or referenced.

(22) Chapter 9, page 11 section 9.5 note 2
What if an entry has OUT parameters but the accept has no statements -- what happens if you use the parameters?

(23) Chapter 9, page 12 section 9.6
Typo in PACKAGE CALENDAR: 2009 should be 2099

(24) Chapter 9, page 13 section 9.6 note 1
Heed your note and make the correction (we would have liked to have seen this explanation).

(25) Chapter 9, page 14 section 9.7.1 paragraph 2, line 1
The line: "A selective wait must contain at least one alternative ... " should read: "A selective wait must contain at least one select alternative ... " since that is the non-terminal used in the syntactic definition.

(26) Chapter 9, page 14 section 9.7.1 paragraph 2
Parenthesized comments in this paragraph specifying the combinations of terminate, delay and else parts allowed in a select statement should not be parenthesized -- they are too important. They should be separately stated and elaborated.

(27) Chapter 9, page 15 section 9.7.1 dashed paragraph 1
When does the delay start? Is it safe for the programmer to assume that the total amount of time required for the select statement if no rendezvous is posssible, is no greater than that given in the delay statement or is the time needed to evaluate any guards not included in the execution time of a select statement?

(28) Chapter 9, page 16 section 9.7.2 paragraph 1, line 1
Use of the word immediately is confusing. A conditional entry call may take an arbitrary amount of (communication) time to execute even if no rendezvous occurs.

(29) Chapter 9, page 17 section 9.7.3
What does the delay include in a timed entry call? Is it the time on the entry queue only, or does it include "message transmission" time to/from caller from/to callee? If the latter, how do we implement this when one task is on Earth and the other on Mars (this is not a flippant question)? Also, if no scheduling algorithm is assumed by the language definition, how can the "correct" execution be guaranteed? We assume a timed entry call with delay 0 really means the programmer is prepared to wait 0 seconds on the entry queue. It is clearly impossible to have any other meaning because an entry call always takes some time. Thus we assume delay 1 means wait for a duration of 1 on the queue. Is this correct?

(30) Chapter 9, page 18 section 9.8 rule
The word "sensibly" is not appropriate in this context. It, is far too ambiguous for a document puporting to be a language definition.

(31) Chapter 9, page 18 section 9.8 last paragraph
If two tasks rendezvous, one with priority 5 and the other without defined priority, is it a valid implementation for the rendezvous to always occur with priority PRIORITY'LAST+1?

(32) Chapter 9, page 20 section 9.10
It is impossible to guarantee that a task named in an ABORT statement will not proceed beyond an accept (etc.) after the aborting task thinks

the aborted task has been marked abnormal. If the tasks were running on different physical processors the communication time for the abort message could be arbitrarily long. Is it legitimate for a task that has been marked abnormal to execute an ABORT statement? The manual implies 'yes'.

(33) Chapter 9, page 20 section 9.10
What happens to the caller/callee in a rendezvous when the callee/caller is aborted? (This is explained in Chapter 11 but should be in section 9.10 also)

(34) Chapter 9, page 21 section 9.11
This section as a whole and the usage of the SHARED_VARIABLE_UPDATE procedure in particular is not at all clear -- we need an example. Also you should point out that use of shared variables makes programs non-portable because this facility may not cover all types in an implementation.

(35) Chapter 9, page 20 section 9.10 (general)
Can the task below be terminated by an ABORT statement once the rendezvous has begun?

```
TASK TRAP IS
   ENTRY X;
END;
TASK BODY TRAP IS
   BEGIN
      ACCEPT X DO
         LOOP
            NULL;
         END LOOP;
      END;
   END;
```